# Interaktive massiv parallele Visualisierung großer Datenmengen aus Strömungssimulationen

Vom Fachbereich Informatik (FB 20)
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor–Ingenieurs (Dr.–Ing.)

von

Dipl.–Math. Sascha Schneider

geboren in Darmstadt

| | |
|---|---|
| Referent: | Prof. Dr.–Ing. Dr. h.c. Dr. E.h. José L. Encarnação,<br>Technische Universität Darmstadt |
| Korreferent: | Prof. Dr. Reinhard Klein,<br>Universität Bonn |
| Tag der Einreichung: | 12.10.2006 |
| Tag der mündlichen Prüfung: | 24.11.2006 |

Darmstädter Dissertation
D 17
Darmstadt, 2006

Widmung

Für meine Familie

# Vorwort

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fraunhofer Institut für Graphische Datenverarbeitung (IGD) in Darmstadt.

Mein besonderer Dank gilt den zahlreichen Personen und Institutionen, die auf unterschiedliche Art und Weise zum erfolgreichen Abschluss dieser Arbeit beigetragen haben. Dazu zählen

- Herr Prof. Dr.–Ing. Dr. h.c. Dr. E.h. José L. Encarnação, dem ich für die Annahme und Betreuung dieser Dissertation und für die gemeinsame Durchführung von Diplomarbeiten danke,
- Herr Prof. Dr. Reinhard Klein, dem ich für sein Interesse und die Bereitschaft zur Übernahme des Koreferats und für die Zusammenarbeit in seiner Zeit als Abteilungsleiter danke,
- Herr Prof. Dr. Volker Luckas, dem ich in seiner Funktion als Abteilungsleiter für sein Vertrauen in meine Arbeit und der langjährigen Zusammenarbeit danke,
- Herr Dr. Jörn Kohlhammer, dem ich für seine Zeit als Abteilungsleiter und sein Vertrauen in mich während der Finalisierung meiner Arbeit danke,
- mein ehemaliger Zimmerkollege Marcus Hoffmann, dem ich für den regen Gedankenaustausch und die produktive Arbeit an gemeinsamen Themen danke,
- meine ehemaligen Kollegen der Abteilung A3 „Echtzeitlösungen für Simulation und Visual Analytics“ (ehemals „Animation und Bildkommunikation“), insbesondere Clemens Gross, Thorsten May, Ingo Soetebier, Jörg Sahm, Arnulph Fuhrmann und Eric Blechschmitt, denen ich für die erfolgreiche gemeinsame Zeit danke,
- meine ehemalige Abteilungs-Sekretärin Gabi Knöß, ohne die die A3 nie so gut funktioniert hätte, wie es der Fall war,
- meine ehemaligen Diplomanden und Hiwis, denen ich für die gute Zusammenarbeit danke,
- meine sonstigen Kollegen am Institut, insbesondere Carola Eichel und Sabine Bartsch, denen ich für die Unterstützung bei der Abwicklung der Promotionsformalitäten danke,

- meine Freunde und Verwandten, die mich während meiner Freizeit erfolgreich von beruflichen Themen ablenkten,
- meine Eltern Heidi und Georg, die mir mein Studium ermöglichten und mich während meiner Forschungsarbeit geistig und moralisch unterstützten,
- und vor allem meine Frau Kerstin und meine Tochter Sonja, die ich über alles liebe.


Der Langsamste, der sein Ziel nicht aus den Augen verliert,
geht noch immer geschwinder, als jener, der ohne Ziel umherirrt.
*Gotthold Ephraim Lessing*

Nicht das Beginnen wird belohnt, sondern einzig und allein das Durchhalten.
*Katharina von Siena*

oder in eigenen Worten:
Am Ende schaffen es diejenigen ins Ziel, die es sich am meisten wünschen.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Heutzutage sind rechnergestützte Simulationen von physikalischen Phänomenen, Prozessen und Vorgängen in Forschung und Industrie zu einem sehr wichtigen Thema geworden. In diesem Zusammenhang ist es seit längerem gängige Praxis, bereits während der Planung von neuen Produkten, z.B. in der Automobil- oder Flugzeugindustrie, die entwickelten Konstruktionsmodelle mit Hilfe komplexer Simulationen zu verifizieren. Auf diese Weise lassen sich bei der anschließenden Fertigung erhebliche Produktionskosten, die durch teure Modelle, aufwendige Prototypen oder Versuchsreihen anfallen, einsparen. In diesem Umfeld sind bereits viele Simulationswerkzeuge entstanden, die sich im Laufe der Zeit im Markt etabliert haben (z.B. [FLU, FEM]). Es ist gängige Praxis, mit Hilfe dieser Werkzeuge beispielsweise Strömungsberechnungen, Hitzeverteilung oder sogar Oberflächen-Verformungen zu simulieren, um die untersuchten Modelle und Objekte dadurch zu verbessern bzw. zu optimieren.

Neben der Industrie und deren produkttechnischen Entwicklungen sind Simulationen komplexer physikalischer Probleme auch für andere Anwender interessant. Die physikalisch basierte Simulation und darauf aufsetzende technisch-wissenschaftliche Visualisierung hat sich zu einem interessanten und wichtigen Forschungsbereich entwickelt, der ein breites Spektrum an möglichen Einsatzgebieten eröffnet. Hierzu gibt es zahlreiche Beispiele. Eines von ihnen, das sowohl für den zivilen, als auch im militärischen Bereich interessant ist, ist die Analyse der Ausbreitung von toxischen Gasen und / oder Krankheitserregern, die gerade in den letzten Jahren stark an Bedeutung gewonnen hat. In diesem Zusammenhang existieren eine Reihe von Forschungsprojekten, die vom BMBF und der EU gefördert werden. Weitere Einsatzszenarien für physikalisch basierte Simulationen ist zum Beispiel die Klimaforschung (Wettervorhersage, Treibhauseffekt, Ozonloch, ...), die virtuelle Nachbildung von chemisch, physikalischen Vorgängen (Gasausbreitung, Verbrennung, Aerodynamik, ...) oder auch die Simulation von Naturkatastrophen (Staudammbruch, Austreten von Schadstoffen in einem Kernkraftwerk, Chemieunfälle, ...). Das ganze ist vor allen Dingen auch dadurch interessant, da man mit Hilfe physikalisch basierter Simulationsmodelle auch Szenarien abbilden kann, die in der Praxis nur unter erheblichen Kosten und Risikien, teilweise sogar überhaupt nicht

durchführbar sind. Gleichzeitig werden die zugrundeliegenden Berechnungsmodelle auch heute noch, meist durch Angleichung an reale Messproben, weiter verbessert.

## 1.2 Problemstellung und Zielsetzung

Bis auf wenige Ausnahmen haben die bisher verfügbaren physikalisch basierten Simulationsapplikationen die Eigenschaft gemeinsam, dass sie numerische Verfahren auf diskretisierten Datenfeldern zur Problemlösung einsetzen. Um nun vorgegebene Ergebnisgenauigkeiten garantieren zu können, müssen sie demzufolge mit sehr großen Datenvolumen arbeiten. Die schiere Menge der erzeugten Ergebnisdaten stellt heutzutage ein Kernproblem für die der Simulation angeschlossenen Visualisierung dar.

Aktuell bekannte Lösungen für dieses Problem gestalten sich auch heute noch sehr mühsam und ineffektiv. Verfügbare Visualisierungsansätze bestehen meist aus kleinen Programmen, die oft direkt in die Simulations-Softwarelösungen integriert sind oder als Zusatzmodule mitgeliefert werden. Auf dem freien Markt existieren nur wenige Softwarelösungen, die völlig unabhängig von den verwendeten Simulatoren oder den modellierten physikalischen Vorgängen arbeiten können [VTK, OPEb]. In der Regel sind die Visualisierungswerkzeuge immer an bestimmte Programme und / oder Datenstrukturen (z.B. verwendete Gittertypen) angepasst [pV3, AMI, VISa]. Allen diesen Ansätzen gemeinsam ist, dass sie aufgrund der internen Struktur nicht mit sehr großen Datenmengen zurechtkommen. Zusätzlich ist meist die Anzahl der unterstützten Daten-Eingabeformate beschränkt.

Hilfreich für die Darstellung der ermittelten Ergebnisdatensätze wäre die Entwicklung eines Werkzeugs, das aufgrund seiner weitestgehenden Unabhängigkeit von den Eingabedaten, auf möglichst vielen physikalisch basierten Simulationsdaten aufsetzen kann. Auf diese Weise wäre es möglich, das Visualisierungswerkzeug und die darin enthaltenen Visualisierungsmethoden auf die unterschiedlichsten Probleme anzuwenden, ohne jeweils immer ein neues speziell angepasstes Visualisierungssystem entwickeln zu müssen.

Durch die Trennung der Visualisierung von der physikalisch basierten Simulation bzw. deren Abkapselung über mächtige, abstrakte Schnittstellen, lassen sich die heute verfügbaren und ausgereiften Simulationswerkzeuge weiterverwenden. Gleichzeitig erhält man jedoch noch einen zusätzlichen Vorteil: Man kann die Visualisierung unabhängig von der Simulationsseite optimieren und weiterentwickeln. Das Potential, das diese Trennung mit sich bringt, ist die Möglichkeit, die Leistungsfähigkeit der Visualisierung durch den Einsatz spezieller Methoden und Algorithmen zu verbessern ohne den speziellen Richtlinien und Vorgaben der Simulationsseite zu unterliegen. Dadurch kann man den enormen Vorsprung in Performanz und Effizienz, den die physikalisch basierte Simulation gegenüber der Visualisierung heutzutage hat, wieder wett machen. Dieses Konzept wird im Rahmen der vorliegenden Arbeit untersucht und prototypisch realisiert.

### 1.2.1 Große Datenmengen und die technisch-wissenschaftliche Visualisierung

Die physikalisch basierte Simulation ist ein effizientes Mittel, um komplexe Sachzusammenhänge im Rechner nachzubilden. Wie bereits beschrieben, erzeugt diese Art von Simulation verfahrensbedingt meist große Datenmengen, die in Form von Dateien abgespeichert werden. Die Schwierigkeit für die wissenschaftlich-technische Visualisierung besteht nun darin, die für sie wichtigen Informationen aus diesen großen Datensätzen zu extrahieren, zu filtern und zu organisieren, um sie anschließend graphisch aufzubereiten. Die Simulationsroutinen an sich, sind durch viele Algorithmen und Optimierungsmethoden sehr weit entwickelt. Um den großen Datenmengen habhaft zu werden bzw. die darin verborgenen komplexen Sachverhalte analysieren zu können, bedient man sich *wissenschaftlicher Visualisierungsmethoden* (vgl. Kapitel 3.6.1). Mit ihrer Hilfe wird es möglich, die Daten sinnvoll zu strukturieren und aufzubereiten, so dass der Anwender die nötigen Erkenntnisse aus den Simulationsvorgängen ziehen kann. Genau in diesem Bereich stoßen allerdings die heute verfügbaren Ansätze zur Lösung des Visualisierungsproblems aufgrund der zu bewältigenden Masse an Simulationsdaten an ihre Leistungsgrenze.

Ein Lösungsansatz für dieses Datenmengenproblem wurde im Forschungsbereich der *visuellen Simulation* [She89] vorgestellt. Hierbei beruhen die verwendeten Formeln und Algorithmen nicht mehr komplett auf den physikalischen Grundlagen. Sie verwenden statt dessen entsprechende mathematische Vereinfachungen. Dadurch sind die ermittelten „Lösungen“ natürlich auch nicht mehr 100% korrekt, wirken aber auf den Betrachter der virtuellen Szene trotzdem „echt“ (Stichwort: *visuelle Korrektheit*). Einige interessante Arbeiten in diesem Bereich findet man beispielsweise in [Sta99, Sak93, Ebe94] und [SKA+00].

Diese Herangehensweise an das Problem ist allerdings nur für einen gewissen Anwenderbereich der Computergraphik interessant. Sie kann immer nur dann eingesetzt werden, wenn es sich lediglich um die Anzeige einer nur näherungsweise präzisen Lösung der Problemstellung handelt, die lediglich visuell korrekt wirkt. Liegt jedoch der Schwerpunkt der Visualisierung auf der möglichst exakten Darstellung der berechneten Simulationsdaten, da beispielsweise ein analysierender Experte daran interessiert ist, möglichst korrekte Details seiner wissenschaftlichen Simulation zu erkennen, so führt diese Methode zu keinem brauchbaren Ergebnis. In diesem Fall ist es nicht ausreichend, die Lösung nur näherungsweise zu ermitteln. Sie muss nicht nur detailgetreu simuliert sondern auch visualisiert werden. Es ist daher notwendig, ein Visualisierungswerkzeug zur Verfügung zu stellen, das auch innerhalb großer Datenmengen den zu analysierenden Datenbereich mit größtmöglicher Genauigkeit darstellen kann.

### 1.2.2 Multivariate Visualisierung

Ein weiterer interessanter Forschungsschwerpunkt in Bezug auf physikalisch basierten Simulationsdaten, ist die multivariate Visualisierung der Datensätze. Multivariat bezeichnet hierbei die Möglichkeit, mehrere Visualisierungstechniken in einer virtuellen Szene beliebig zu kombinieren. Wichtig hierbei ist die notwendige Unterscheidung zwischen *wissenschaftlich technischer Visualisierung* auf der einen Seite und *realitätsnaher Visualisierung* auf der

anderen Seite: So kann die Art der darzustellenden Daten alleine schon Einschränkungen in den darauf basierenden sinnvollen Visualisierungsarten bedeuten. Zum Beispiel ist eine Darstellung von Simulationsdaten über die Ausbreitung von Krankheitserregern nur schwer möglich, ohne sich wissenschaftlicher Visualisierungsmethoden (Falschfarben, Gebietseinfärbungen, Volumenrendering etc.) zu bedienen, da sie als mikroskopisch kleine Partikel in einer realitätsnahen Visualisierung nicht erkennbar wären.

An dieser Stelle stößt man heutzutage auf ein weiteres Kernproblem aktueller Visualisierungsansätze: Verfügbare Visualisierungssysteme integrieren meist nur eine limitierte Teilmenge aktuell bekannter und etablierter Visualiserungsmethoden oder sind nur begrenzt um neue erweiterbar. Zusätzlich wird der Benutzer bei der Auswahl der Visualisierungsmethoden oft darin beschränkt, welche er von ihnen gleichzeitig in der Szene aktivieren kann.

Hilfreich wäre ein System, welches es dem Anwender einerseits erlaubt, schnell neue Visualisierungsmethoden in das bestehende System zu integrieren und andererseits beliebige Kombinationen aus allen verfügbaren Visualisierungsmethoden gleichzeitig in einer virtuellen Szene ineinander blenden zu können. Es sollte gewissermaßen möglich sein, bei der Wahl der Darstellung beliebig aus einem Fundus an vorhandenen Visualisierungsmethoden schöpfen zu können, der zudem einfach erweiterbar ist. Um bei dem oben beschriebenen Beispiel der Ausbreitung von Krankheitserregern zu blieben, sollte es beispielsweise erlaubt sein, realistisch gerenderte 3D Landschaftsgeometrieen mit einer oder mehreren wissenschaftlich technischen Visualisierungsmethoden (z.B. Strömungslinien, vgl. Kapitel 3.5.1.3) für die Erregerausbreitung in einer Szene zu kombinieren.

Aus diesem Grund ist es ein weiteres Ziel dieser Arbeit, zu untersuchen, inwieweit es möglich ist, verschiedene realistische und wissenschaftlich technische Visualisierungsverfahren innerhalb einer Szene zu kombinieren und gleichzeitig darzustellen. Durch die Integration dieser Fähigkeit in ein Visualisierungsdaten-Analysewerkzeug, kann dem Benutzer viel Arbeit bei der Analyse und Interpretation, der auf die Datenmenge aufsetzenden Visualisierung, abgenommen werden. Auf lange Sicht wäre es dadurch sogar möglich, auch dem unerfahrenen Anwender visuellen Zugang zu den Simulationsergebnissen zu gewähren, da sich dabei auch realistische Visualisierungsmethoden mit wissenschaftlich technischen Darstellungen beliebig mischen lassen. Das Verständnis der dargestellten Sachverhalte wird somit unterstützt.

Zusammenfassend konzentriert sich die vorliegende Arbeit im wesentlichen auf folgende drei Anforderungen:

- Die Visualisierung physikalisch basierter Simulationsdaten soll von der eigentlichen Simulation entkoppelt werden, damit sie separat von ihr weiterentwickelt werden kann.
- Es muß eine Visualisierungs-Technologie entwickelt werden, die mit der Leistungsfähigkeit aktueller Simulationsumgebungen für physikalische Strömungsprobleme mithalten kann. Dies beinhaltet die Unterstützung großer Simulationsdatenmengen.
- Die Wahl geeigneter Visualisierungsmethoden und ihrer Parametereinstellungen soll so gestaltet werden, dass es möglich wird beliebige Kombinationen von ihnen gleichzeitig in einer Szene darzustellen.

## 1.3 Gliederung

In der vorliegenden Arbeit wird ein neuer allgemeiner Ansatz für die Visualisierung physikalisch basierter, wissenschaftlicher Simulationsdaten vorgestellt, der in der Lage ist, mit den darzustellenden großen Datenmengen effizient und schnell umzugehen und diese dem Anwender durch möglichst effizienten und automatisierten Zugang nahe zu bringen.

In Kapitel 2 erfolgt eine Vorstellung der Einsatzmöglichkeiten und Anwendungsgebiete. Hier wird eine Übersicht gegeben, in welchem unterschiedlichen Themenbereichen Strömungsvisualisierung Verwendung findet.

Das daran anschließende Kapitel 3 beinhaltet eine Einordnung der Thematik in den wissenschaftlichen Gesamtzusammenhang. Zusätzlich werden bereits verfügbare Visualisierungswerkzeuge und deren ansatzbedingte Probleme genauer untersucht und diskutiert. Darüber hinaus werden die zum Verständnis dieser Arbeit notwendigen Grundlagen für die Entwicklung eines Lösungsansatzes zum Problem der Visualisierung großer Datenmengen vorgestellt.

Kapitel 4 formuliert zunächst eine Analyse der Problemstellung. Anhand der daraus gewonnenen Erkenntnisse werden Anforderungen an einen neuen Visualisierungstechnologie-Ansatz für Stömungssimulationsdaten entwickelt.

Darauf aufbauend wird in Kapitel 5 ein Konzept zur Lösung der Problematik „Visualisierung großer Strömungsdatenmengen“ aus physikalisch basierten Simulationen entwickelt.

In Kapitel 6 erfolgt die genaue Spezifikation der im Lösungsansatz entwickelten Begriffe und Algorithmen.

Gegenstand von Kapitel 7 ist die prototypische Implementierung des in den beiden vorangegangenen Kapiteln verfolgten Lösungsansatzes, der anhand von Anwendungs-Szenarien vorgestellt und präsentiert wird. Einige ausgewählte Visualisierungsmethoden werden genauer vorgestellt.

Die vorliegende Arbeit schließt in Kapitel 8 mit einer Zusammenfassung der Ergebnisse und deren Relevanz im Hinblick auf zukünftige Entwicklungen.

# Kapitel 2

# Einsatzmöglichkeiten und Anwendungsgebiete

Die Visualisierung von Strömungsdaten wird heutzutage in den unterschiedlichsten Anwendungsgebieten eingesetzt. Vorrangig zu nennen ist hierbei der Bereich der technisch-wissenschaftlichen Visualisierung, die Kunst und Unterhaltungsindustrie sowie mehr und mehr auch die Computerspieleindustrie. Typische Aufgabenstellungen sind die Präsentation abstrakter Sachzusammenhänge oder die wirklichkeitsgetreue Abbildung der Realität. Die Anforderungen und Zielsetzungen der jeweiligen Anwendungsgebiete unterscheiden sich hierbei ebenso deutlich, wie die zugehörigen Strategien zur Problemlösung. Zwar wird im technisch-wissenschaftlichen Bereich Strömungsdatenvisualisierung seit längerer Zeit erfolgreich eingesetzt, jedoch gibt es auch hier die unterschiedlichsten Probleme, die meist ansatzbedingt sind. Ganz anders sieht es in der Kunst und Unterhaltungsbranche aus. Hier wird meist ungeachtet des finanziellen und personellen Aufwandes der Einsatz von Strömungsdaten bzw. deren qualitativ hocherwertige Visualisierung massiv vorangetrieben. Der Fokus liegt dabei hauptsächlich auf dem optischen Ergebnis. Der erforderliche Aufwand wird als notwendig akzeptiert.

Die Gründe für die Ausbreitung der Strömungssimulation und der zugehörigen Visualisierung auf mittlerweile sehr vielen unterschiedichen Gebieten sind vielseitig. Neben der gestiegenen Leistungsfähigkeit entsprechender Systeme, verursacht durch die ständige Weiterentwicklung der zur Verfügung stehenden Hardware und Software, ist es vor allen Dingen der vereinfachte Zugang zu entsprechenden Simulations- und Visualisierungswerkzeugen, der die Anwendbarkeit bzw. Übertragbarkeit auf viele verschiedene Bereiche fördert. War es früher noch die Aufgabe von wenigen Spezialisten entsprechende Modelle im Rechner abzubilden, zu steuern und zu interpretieren, so sind heutzutage fertige Softwarepakete auf dem Markt verfügbar, die komplexe physikalische Strömungszusammenhänge abbilden und simulieren können. Durch den bequemen Zugang zu dieser Technik und dem erleichterten Umgang mit ihr, breiten sich die entsprechenden Techniken immer weiter in die verschiedensten Bereiche aus.

Die prinzipielle Vorgehensweise in allen hier vorgestellten Bereichen beruht dabei auf demselben Prinzip: In einer vorgeschalteten physikalisch basierten Simulation werden Strömungs-

daten berechnet, die im Anschluß ihren Weg in die dargestellte Szene finden. Die Simulation geht dabei immer Hand in Hand mit der zugehörigen Visualisierung.

Betrachtet man Anforderungen und Lösungswege der unterschiedlichen Bereiche im Detail, so finden sich sehr schnell Ansatzpunkte, die die Einführung eines alternativen Konzeptes zur massiv parallelen Visualisierung von großen Strömungsdatenmengen rechtfertigen. In diesem Kapitel wird eine Übersicht über die unterschiedlichen Einsatzmöglichkeiten und Anwendungsgebiete der Strömungsdatenvisualisierung gegeben.

## 2.1 Technisch-wissenschaftliche Visualisierung

Der Bereich der technisch-wissenschaftlichen Strömungsdatenvisualisierung ist zugleich der „älteste“, als auch der am weitesten entwickelte. Seit Beginn der physikalisch basierten Strömungssimulation wurde schon immer mit Hilfe von technisch-wissenschaftlichen Visualisierungskonzepten die generierte Datenmenge analysiert und versucht, interpretierbare wissenschaftliche Darstellungen aus der Masse der Daten zu extrahieren.

Die klassischen Anwendungsgebiete für die Strömungsdatenvisualisierung - speziell auch großer Strömungsdatenmengen - zerfallen in eine Vielzahl unterschiedlicher Bereiche, wobei die hier vorgestellte Liste nur einen kleinen Teil der tatsächlichen Anwendungsgebiete umfasst:

**Automobil-, Schiffahrt-, Flugzeug- und Raumfahrtindustrie**

Seit dem Einzug der Computertechnologie in diese Bereiche der Fertigungsindustrie, wurde das Einsatzgebiet der Strömungssimulation hier immer weiter ausgebaut. Standen zu Anfangs lediglich Techniken zur Verfügung, die der Automobilindustrie den „virtuellen Windkanal“ [MTRB01] zur Verfügung stellten, so werden heutzutage dank präziserer Modelle und der zur Verfügung stehenden Rechnerkapazität bereits hochkomplexe Simulationen wie beispielsweise Verbrennungsverläufe im Motor mit unterschiedlichen Gas / Benzin Mischverhältnissen, deren Abströmverhalten oder sogar die Frischluftzirkulation in der Fahrerkabine untersucht.

Auch in der Flugzeugindustrie und Raumfahrt hat sich das Einsatzgebiet der Strömungssimulation im Verlauf der Zeit immer weiter gewandelt bzw. ausgedehnt. Wurden anfänglich mit ihrer Hilfe lediglich Untersuchungen der Luftumströmung der Karosserie getätigt oder das Auftriebverhalten der eingesetzten Tragflächenprofile bestimmt, so wird auch hier mittlerweile wesentlich mehr von der Simulation erfaßt. War es früher aufgrund des sich dahinter verbergenden Rechenaufwandes noch so gut wie undenkbar, Propellermotoren mit in den zur Verfügung stehenden Mitteln im Rechner abzubilden, so besteht heute bereits die Möglichkeit, ganze Düsentriebwerke, bestehend aus einigen hundert Schaufelblättern, zu simulieren und zu optimieren.

Klassische Einsatzgebiete der Strömungssimulation in der Schiffahrt ist die Konstruktion eines Schiffsrumpfs. Hier steht deren Verhalten im Wasser, d.h. deren Strömungsverhalten im Vordergrund.

Neben den Schwerpunkten, die die Flugzeugindustrie mit sich bringt, kommen im Gebiet der Raumfahrt noch weitere Einsatzgebiete für die Strömungssimulation hinzu. Neben der Simulation beispielsweise des Wiedereintrittes der Raumfähre in die Atmosphäre werden heutzutage auch die zugehörigen Raketentriebwerke mit Hilfe der Strömungssimulation und Visualisierung immer weiterentwickelt.

Heutzutage ist die Fertigungsindustrie fest mit dem Thema der Simulation und Visualisierung von Strömungsdaten verwachsen. Diese beiden Techniken haben sich zu einem festen Werkzeug und Bestandteil dieser Industrie entwickelt. Eine entsprechende effiziente Vorgehensweise ohne den Einsatz dieser Techniken ist nicht mehr denkbar.

**Klimaforschung, Umweltschutz**

Klassisches Einsatzgebiet der Strömungssimulation und Visualisierung im Bereich der Klimaforschung ist die Wettervorhersage. Hier werden mit Hilfe von sehr komplexen Modellen die Wetterverhältnisse für einen definierten Zeitraum berechnet und ausgewertet. Hierzu gehören auch Extreme wie beispielsweise das Verhalten von Wirbelstürmen bzw. deren Verlauf und die Vorberechnung des Weges, den diese einschlagen werden.

Darüber hinaus werden auch Langzeituntersuchungen durchgeführt, die in den Bereich der Klimaforschung fallen. Prominentestes Beispiel hier ist die Untersuchung des Ozonlochs und Prognosen über seine Ausbreitung in der Zukunft. Auch Untersuchungen zum Abschmelzen der Polkappen bzw. die Auswirkungen auf das Weltklima und beispielsweise den Meerespiegel fallen in diesen Bereich. Außerdem wären hier auch die Untersuchungen der Meeresströmungen und deren Auswirkungen das Klima zu erwähnen.

**Sicherheitsuntersuchungen, Katastrophenschutz**

Auch auf dem Gebiet der Sicherheit und des Katastrophenschutzes findet die Strömungssimulation und Visualisierung mehr und mehr Anwendung. Auch hier sind unterschiedliche Einsatzgebiete zu unterscheiden, die jedoch alle mit dem Thema der Strömungsdaten zusammenhängen. In der Regel dienen die in diesem Themenbereich durchgeführten Untersuchungen der Sicherheit und der Vorbeugung bzw. Vermeidung von Unfällen oder Katastrophen.

Generell gibt es auch hier zahlreiche Möglichkeiten für den Einsatz der Strömungssimulation und der zugehörigen technisch-wissenschaftlichen Visualisierung. Eines der Themen, die in diesem Zusammenhang zu nennen sind, ist beispielsweise die Simulation des Ausbreitens schädlicher Substanzen wie etwa Ölteppiche, verursacht von undichten Pipelines, Ölplattformen oder Tankern. Auch das Einsickern von Schadstoffen in den Untergrund oder deren Verbreitung im Trinkwasser gehören zu diesem Themengebiet, genauso wie die Simulation des Ausbreitungsverhaltens von luftgebundenen Krankheitserregern oder von giftigen Gasen nach Chemie-Unfällen.

Neben diesen Szenerien wird die vorgestellte Technik auch für die Materialprüfung verwendet. So wird beispielsweise Beton auf seine Hitzebeständigkeit untersucht [hoc05], außerdem das Ausbreiten von Feuern und Rauch, z.B. in Tunneln [VIR] und Gebäuden. Ein weiteres offenes Forschungsgebiet ist die Simulation von Flächen- bzw. Waldbrän-

den [wal06] unter unterschiedlichen Randbedingungen und insbesondere deren Visualisierung.

Außer den hier vorgestellten Bereichen gibt es mittlerweile noch eine Vielzahl weiterer Gebiete, auf denen die Strömungssimulation und speziell die zugehörige Visualisierung zum Einsatz kommt. Das Spektrum reicht von der Glasherstellung bzw. der Entwicklung des zugehörigen Brennofens, bis hin zur Simulation von Hochdruck Ultraschall Schockwellen und deren Auswirkungen auf verschiedene Materialien [COS05]. Die wenigen hier vorgestellten Beispiele sollen lediglich zur Anschauung dienen, wie verbreitet die technisch-wissenschaftliche Strömungsdatenvisualisierung mittlerweile ist bzw. wie wichtig sie inzwischen für die unterschiedlichsten Bereiche geworden ist.

Die Gründe, die zum Einsatz der technisch-wissenschaftlichen Visualisierung in den hier vorgestellten Bereichen führen, lassen sich grob zu Schwerpunkten zusammenfassen, die in den beiden folgenden Unterkapiteln näher beleuchtet werden.

### 2.1.1 Forschung: Modellbildung und Analyse

In diesem Bereich zerfallen die Anwendungsgebiete in verschiedene Kategorien, die hier kurz aufgelistet sind.

**Datenanalyse**
: In der Datenanalyse werden bestimmte vorgegebene Szenarien untersucht und deren Sachzusammenhang in einem entsprechenden Modell abgebildet. Dieses Modell wird mit einer Vielzahl unterschiedlicher Parameter variiert, um unterschiedliche Rahmenbedingungen und Ausgangssituationen für die dahinter verborgene Simulation zu generieren. Die Daten der unterschiedlichen Szenarien werden gesammelt und in einer Analysephase im Anschluß visualisiert und untersucht. Grund für den Einsatz der Strömungsvisualisierung ist die Untersuchung eines vorgegebenen Modells bei unterschiedlichen Voraussetzungen (beispielsweise das Verhalten eines Ventilationssystems in einem Tunnel bei unterschiedlichen Luftdruckverhältnissen auf beiden Seiten des umgebenden Berges). Hier kann die technisch-wissenschaftliche Visualisierung helfen, die komplexen Sachzusammenhänge zu verstehen und deren Verhalten zu analysieren.

**Konstruktion + Design**
: Im Bereich der Konstruktion und des Designs wird die Strömungsdatenvisualisierung in der Regel in der Planungsphase des untersuchten Gegenstandes eingesetzt. Hier wird entschieden, ob sich die Konstruktion des untersuchten Objektes lohnt und das Design stimmig ist. Aufgrund der Ergebnisse der Visualisierung wird dann entschieden, wie weiter im Produktionsprozess verfahren werden soll, d.h. ob bespielsweise das aktuelle Konzept verworfen und ein neues geplant werden muss, oder ob das vorgestellte Design korrekt und funktionell ist. Diese Phase des computerunterstützen Konstruierens und Designens im Produktionsprozess hat sich mitterweile in vielen Marktbereichen etabliert und ist dort zu einem wichtigen Bestandteil der Fertigung geworden.

**Modellprüfung / -entwicklung**
Die Modellprüfung und -entwicklung beschreibt den Bereich, der sich mit dem Entwurf neuer Berechnungsmodelle für bestimmte untersuchte Phänomene beschäftigt. Viele unterschiedliche Einsatzgebiete der Strömungssimulation benötigen ganz spezifische Rechenmodelle, die die dort vorherrschenden physikalischen Bedingungen möglichst exakt auf mathematische Formeln abbilden. Diese Formeln müssen speziell entwickelt und die erforderlichen Steuerparameter und Konstanten bestimmt und in die Berechnung eingesteuert werden. In dieser Phase sind meist mehrere Zyklen nötig, die darüber entscheiden, ob ein vorgestelltes Simulationsmodell die physikalische Wirklichkeit im untersuchten Einzelfall korrekt abbildet. Zur Verifikation der eingesetzten Berechnungen werden meist deren visualsierte Ergebnisse mit den Messewerten aus der Realität direkt verglichen und die Modelle entsprechend angepaßt.

**Validierung, Test**
Die Validierung und Testphase beschreibt den Bereich, in dem es darum geht, fertige Produkte oder Gegenstände auf deren Tauglichkeit unter vorgegebenen Richtlichen zu überprüfen. Hier wird beispielsweise untersucht, ob das verwendete Material bestimmten vorgegebenen Belastungen standhalten kann (z.B. ob eine der Hitze ausgesetzte Eisenstange sich unter einer bestimmten Last durchbiegt oder nicht) oder das entwickelte Modell die vorgegebenen Randbedingungen erfüllt (z.B. Luftwiderstand einer Autokarosserie bei einer vorgegebenen Geschwindikeit). Die fertigen Objekte werden gegen vorgegebene Rahmenbedingungen validiert. In dieser Phase wird auch darüber entschieden, ob bereits existierende Objekte noch den für sie vorgesehenen Aufgaben unter Einhaltung vorgegebener Richtlinien nachkommen können (beispielsweise die Entscheidung darüber, ob ein in die Jahre gekommener Staudamm noch dem Druck und der Durchströmkraft des dahinterliegenden Wassers stand hält). Ein entsprechender Test entscheidet darüber, ob das untersuchte Objekt seiner Aufgabe noch zuverlässig nachkommen kann, oder ausgetauscht werden muss.

### 2.1.2 Präsentation und Verifikation

Neben der Forschung, die sich vorranging mit der Modellbildung und Analyse beschäftigt, ist die Präsentation und Verifikation ein weiteres wichtiges Einsatzgebiet für die technisch-wissenschaftliche Strömungsdatenvisualisierung. Im Gegensatz zur Forschung werden hier auch „Nicht Experten“ mit den Visualisierungsergebnissen konfronziert.

Bei der Präsentation wird ein geplantes Objekt dem Entscheidungsträger oder einem interessierten Personenkreis vorgeführt. Gegebenenfalls wird aufgrund der Präsentation im Anschluß darüber entschieden, wie weiter verfahren werden soll. Hier kann es vorkommen, dass die Personen, denen die Präsentation vorgeführt wird, nicht direkt an deren Entwicklung beteiligt waren und demzufolge auch nicht unbedingt entsprechende Fachkenntnisse haben müssen. Dennoch können sie beispielsweise Entscheidungsträger sein. Die Aufgabe der Präsentation bzw. Visualisierung ist es nun, sie dabei zu unterstützen, einen entsprechenden Beschluss zur weiteren Fertigung oder zum Kauf des vorgestellten Produktes zu tätigen.

Ein Beispiel wäre hierfür der Firmenchef, der darüber entscheiden muss, ob ein von seinem Forschungsstab entwickelts Produkt zur Serienreife geführt oder verworfen wird.

Auch bei der Verifikation geht es darum, ein untersuchtes Objekt einem unter Umständen nicht direkt mit dem Thema befaßten Personenkreis näherzubringen, der dann als „neutrale Instanz“ darüber entscheidet, ob es vorgegebene Richtlinien einhält und damit zum Betrieb zugelassen wird oder nicht. Auch hier kann es vorkommen, dass die Zielgruppe nicht direkten Einblick in die dahinterstehenden Modelle und Verfahren hat, aber dennoch aufgrund der Ergebnisse der Visualisierung über den weiteren Verbleib des vorgestellten Objektes entscheiden muss. Denkbares Beispiel wäre an dieser Stelle eine Art „Tüv“, der die Abnahme eines neuen Produktes mit aufgrund einer ihm vorgestellten Strömungssimulation und Visualisierung durchführt.

Die Aufgabe für die Visualisierung ist in diesem Gebiet also ungleich schwerer, als auf dem Gebiet der reinen Forschung. Hier müssen auch weniger technisch versierte Betrachter bzw. Fachfremde die dargestellten Informationen verstehen und interpretieren können. Eine entsprechende Qualität und Aussagekraft der Darstellung ist somit entscheidend für die Berwertung des Produktes.

## 2.2 Kunst und Unterhaltung

Als George Lucas 1975 die Firma Industrial Light Magic für den Kinofilme Star Wars gründete, wurde für die Filmindustrie ein neues Zeitalter eingeleitet. Die Computertechnologie hielt Einzug in die Filmbranche. War es zuvor noch eine Art echtes „Handwerk“, visuelle Effekte in die Kunst, Film und Fernsehbranche mit Hilfe der unterschiedlichsten optischen Tricks in Filme zu integrieren, so wurde ab diesem Zeitpunkt das Zeitalter der „Special Effects“ eingeleitet: Die 3D Computergraphik erreichte die Filmindustrie. „Virtuelle“ Effekte wurden zu bestehenden echten Aufnahmen hinzugerechnet. Es war nicht länger nötig, die gewünschten optischen Effekte per „Handarbeit“ nachzubilden, sofern das überhaupt möglich war. Man konnte sie einfach im Rechner generieren und dem fertigen Film hinzufügen.

Im Verlauf der Zeit sprossen hierfür vermehrt Firmen und Technologien aus dem Boden. Der entsprechende Marktsektor ist im Verlaufe der Zeit sehr stark gewachsen und zu einem festen Bestandteil der Filmindustrie geworden. Ein Gesamtüberblick über diesen Marktsektor zu erhalten ist heutzutage nur noch schwer möglich, zu zahlreich sind die entsprechenden Anbieter und zur Verfügung stehenden Produkte geworden. Viele Schulen und Universitäten sind bereits dazu übergegangen, entsprechende Ausbildungsberufe genau für diesen Sektor anzubieten. Im Zuge der fortschreitenden Integration der entsprechenden Technologien in den Bereich der Filmindustrie, hat sich auch der zugehörige Arbeitsmarkt entsprechend entwickelt und ist ein wichtiger Bestandteil der heutigen Welt geworden.

Durch den wachsenden Markt ist die Anwendung entspechender Visualisierungs-Technologien deutlich im Preis gefallen. Die Senkung der mit der Integration von Special Effects verbundenen Kosten in einen Kinofilm und die Vereinfachung des Zugangs zu diesen Verfahren bedingt, dass mittlerweile sogar immer weniger Kinofilme erscheinen, die nicht darauf zurückgreifen. War es früher der klare Schwerpunkt von Sciene Fiction oder Fantasy Filmen,

Computergraphik zu integrieren, so geht die Tendenz der Filmindustrie so weit, dass bereits heute in einem ganz normalen Kinofilm mehr Special Effects zum Einsatz kommen, als im klassischen Science Fiction Epos Star Wars von 1975.

Vorrangig wird die Strömungssimulation und deren Darstellung im Zusammenhang mit Filmen für die realitätsnahe Visualisierung genutzt, d.h. die dargestellten Effekte sollen möglichst „echt“ und damit realistisch wirken. Zu nennen sind hier in erster Linie die Darstellung von Feuer und Rauch, beispielsweise bei Explosionsszenen. Diese Art von „Special Effekt“ hielt mit als erste Einzug in die Filmindustrie. Heutzutage werden viele weitere Effekte mit dem Rechner nachgebildet und in den produzierten Film integriert. Zu nennen sind hier beispielsweise simulierte Wettereffekte wie Wolken, Regen oder Schnee. Weitere Beispiele sind simulierter Wind, der Blätter, Haare oder Kleidung bewegt oder erhitzte, Schlieren bildende Luft. Die Darstellung von Wasser, Wellen und Gischt, ein Staudammbruch oder die Darstellung von einem Vulkanausbruch samt zugehörigen pyroklastischen Wolken, Ascheregen und Lava sind mitterweile alle mit Hilfe des Rechners möglich. Selbst deutsche Produktionen müssen hierbei den Vergleich zum wesentlich größeren Markt in Hollywood nicht mehr scheuen.

War man früher, meist aus Kosten- und/oder Hardwareleistungsgründen noch darauf angewiesen, nur Teile des Films mit Visual Effects auszustatten, so geht man mittlerweile sogar dazu über, komplette Filme aus dem Rechner stammen zu lassen. In der Vergangenheit war der Einsatz von Computeranimationen aufgrund des damit verbundenen Aufwandes oft auf wenige kurze Szenen beschränkt. Man fand sie vereinzelt auf verschiedenen Graphik-Fachkonferenzen. Heutzutage ist man bereits dazu übergegangen ganze Kinofilme komplett aus dem Rechner stammen zu lassen. Hier war *Toy Story* 1995, produziert von den Walt Disney Pictures und den Pixar Studios der Vorreiter. Er war der erste voll computergenerierte Kinofilm. Mittlerweile folgten ihm zahlreiche weitere. Zu den aktuelleren Vertretern dieses Genres gehören Filme wie *Ice Age* (Produziert von 20th Century Fox), *Shrek 1+2* oder auch *Findet Nemo*. Auch hier müssen neben der reinen Darstellung der Akteure und der Umgebung (Licht, Schatten, Bewegung) auch die entsprechenden Wetter und Umweltbedinungen realitätsnah und damit glaubhaft visualisiert werden. Mittlerweile sind beispielsweise die Disney Studios dazu übergegangen, die klassische Zeichentrick-Kinofilmproduktion komplett einzustellen und statt dessen 3D Computergrafik basierte Filme zu produzieren.

## 2.3 Computerspieleindustrie

Mit der steigenden Leistungsfähigkeit der zur Verfügung stehenden Hardware, insbesondere der Graphikhardware, erlebt auch die Integration von Strömugssimulationen bzw. der zugehörigen Visualisierung im Bereich der Computerspiele eine starke Vorwärtsentwicklung. Aufgrund der stark limitierten vorhandenen Systemresourcen im Vergleich zur Anwendung in der Forschung, ist dieser Bereich der Computergrafik mit einer der herausfordernsten, in den diese Thematik Einzug hält. Die Leistungsfähigkeit der Hardware, die einem normalen Anwender in Form eines gewöhnlichen Personal Computers (PC) heutzutage zur Verfügung steht, ist ungleich viel größer und effektiver, als es noch vor einigen Jahren der Fall war. Aufgrund der fehlenden Rechenleistung war es früher in der Compouterspieleindustrie üblich, auf (physikalisch basierte) Simulationen und entsprechende Visualisierungen, die über die pure Kernhandlung des Spiels hinausgehen, weitestgehend zu verzichten oder sie auf einfache Art und Weise halbwegs glaubhaft nachzubilden. War früher aufgrund dieser Tatsache eher optisch korrekt wirkendes „Schmummeln“ angesagt, beispielsweise mit vorberechneten Animationssequenzen, so beruht die Darstellung heutzutage immer mehr auf konkreten physikalisch basierten Berechnungsmodellen, die zur Laufzeit ausgeführt werden.

Im Bereich der Computerspiele steht, ähnlich wie in der Filmindustrie, vorangig die realitätsnahe Visualisierung von Strömungssituationen im Vordergrund. Hier ist weniger die technisch-wissenschaftliche Darstellung gefragt, als ein Vermitteln von Umwelteindrücken, die möglichst nahe an die abgebildete und im Spiel simulierte Realität herankommen. Im Zusammenhang mit Computerspielen spricht man hier, gerade bei 3D Graphik Spielen, auch von der Leistungsfähigkeit der im Spiel verwendeten „Physics Engine“. Diese bildet, neben Themen wie realitätsnaher Beleuchtungsrechung oder Kollisionserkennung verschiedener bewegter Objekte mit dem daraus folgenden Abprallverhalten, auch die Simulation von Strömungen und deren Einfluß auf die dargestellte Szene ab. Obwohl sich auch in dem Bereich der Strömungssimulation und Visualisierung in Computerspielen in letzter Zeit sehr viel getan hat, ist sie im Vergleich zur Filmindustrie immer noch ein ganzes Stück in der Entwicklung hinter dieser zurück. Die im direkten Vergleich eingeschränkten Hardwareresourcen, die der Spielesoftware beim Betrieb auf einem PC zur Verfügung stehen, sind ungleich geringer, als die, die bei der Entwicklung eines Filmes verwendet werden können. Aus diesem Grund werden hier auch heute noch vereinfachte und damit rechenbeschleunigte Modelle der abgebildeten Physik verwendet, wenn auch der Trend deutlich hin zu einer weiter reichenden Integration immer komplexerer Strömungsmodelle geht.

Die mittlerweile in Computerspiele integrierten Techniken mit Bezug auf die Strömungsvisualisierung sind dennoch bereits sehr vielseitig, das Thema entsprechend umfangreich. So wird mittlerweile das Verhalten von Wasser, beispielsweise entstehende Spiegelungen und Wellen beim Berühren der Wasseroberfläche, genau wie optische Verzerrungseffekte, die unter Wasser entstehen, im Spiel nachgebildet.

Darüber hinaus werden Dinge wie Explosionen oder das Verhalten von Nebel, der von einem schnellen Objekt durchflogen oder von einem Spieler durchschritten wird, nachgebildet, genauso wie die Visualisierung von Feuerstellen und das Schlierenverhalten der darüber aufsteigenden heißen Luft.

Generell läßt sich sagen, das gerade der Bereich der Computerspieleindustrie ein starker Motor für das Voranschreiten der Qualität und Leistungsfähigkeit der Strömungsvisualisierung auch auf leistungsschwächeren Endgeräten wie einem PC immer weiter vorangetrieben wird. Die ständig weiterwachsende Rechenleistung der zugehörigen Graphikkarten und Prozessoren hält auch in Zukunft noch ein großes Wachstumspotential in diesem Bereich vor.

## 2.4 Zusammenfassung

Gegenstand dieses Kapitels ist die Vorstellung der Einsatzmöglichkeiten und Anwendungsgebiete der Strömungsvisualisierung. Hierbei reicht das Spektrum von der klassischen Forschunganwendung bis hin zur hochwertigen Produktpräsentation. Einzelne Anwendungsfälle werden vorgestellt.

Auf den Bereich der Kunst und Unterhaltung wird genauer eingegangen. Gerade in der Filmindustrie sind in den letzten Jahren viele weitere Einsatzgebiete für die Strömungsvisualisierung hinzugekommen und neue entstanden. Hier steht eine qualitativ hochwertige und realitätsnahe Visualisierung im Vordergrund. Die zugehörige Industrie sowie die dabei zum Einsatz kommenden Techniken befinden sich im Wachstum und fortschreitender Entwicklung.

Schließlich wird noch auf den Bereich der Computerspieleindustrie eingegangen. Gerade dieses Marktsegment ist für die deutliche Vorwärtsentwicklung der Strömungsvisualisierung auch auf kleineren Harwareplattformen wie dem PC mit verantwortlich. Viele moderne Spiele integrieren die Strömungssimulation und Visualisierung in den Spielverlauf. Hier entstehen viele Innovationen und Methoden, die auch für die weitere Verbreitung der Thematik sorgen.

# Kapitel 3

# Grundlagen, existierende Ansätze und Konzepte

In diesem Kapitel werden die theoretischen Grundlagen und Konzepte erläutert, die zum Verstehen der vorliegenden Arbeit nötig sind. Zuerst wird der Begriff der Visualisierungsmethode eingeführt und seine Verwendung im Zusammenhang mit dieser Arbeit definiert. Danach erfolgt eine wissenschaftliche Einordnung des hier vorgestellten Themas in existierende Ansätze und Konzepte. Anschließend wird auf das Thema der parallelen Datenverarbeitung und Visualisierung eingegangen, sowie auf spezielle Gundlagen, die zum Verständnis dieser Arbeit wichtig sind.

## 3.1 Begriffsbildung Visualisierungsmethode

Die physikalisch basierte Simulation von Strömungen, auch *Computational Fluid Dynamics* (*CFD*) genannt, ist, genau wie die darauf aufbauende Visualisierung, ein sehr weitläufiges Gebiet mit einer Vielzahl eingesetzter unterschiedlicher Verfahren und Modelle. Allen Strömungsmodellen gemeinsam ist ihr Ursprung, die Navier Stokes Differentialgleichung [nav05]:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{u} + \frac{\vec{f}}{\rho} \tag{3.1}$$

mit $\nabla \cdot \vec{u} = 0$. Hierbei steht $\vec{u}$ für den Ort, $t$ für die Zeit, $p$ für den Druck, $\rho$ für die Dichte, $\nu$ für die kinematische Viskosität und $\vec{f}$ für die Kraft.

Sie beschreibt das Verhalten von Gasen und Flüssigkeiten. Die unterschiedlichen Strömungsmodelle und -konzepte, die heutzutage verfügbar sind, versuchen alle die Lösung dieser Differentialgleichungen durch unterschiedlichste Annahmen, Verfahren und Methoden zu approximieren - in der Regel angepaßt an die speziellen Gegebenheiten, die im Einzelnen untersucht werden. Der damit verbundene algorithmische Ablauf ist bei all diesen eingesetzten Architekturen ähnlich (siehe Abbildung 3.1):

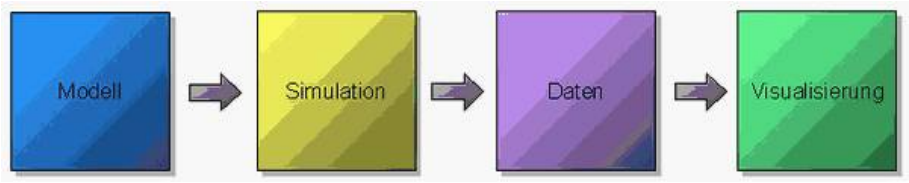



***Abbildung 3.1:*** *Der Datenverlauf von der Modellbildung bis zur Visualisierung*

Zuerst wird ein Berechnungsmodell konstruiert, das der eigentlichen Simulation zugrunde gelegt wird. Diese Simulation errechnet nun für einen bestimmten Sachverhalt das Verhalten der untersuchten Strömung. Hierbei kommen unterschiedlichste Näherungsverfahren für die oben beschriebene Differentialgleichung zum Einsatz. Diese Verfahren diskretisieren den Raum in verschiedene Zellen oder auch Gitterpunkte für die sie aufgrund der verwendeten Lösungsvariante die entsprechenden Ergebnisse berechnen und abspeichern (siehe auch Kapitel 3.3). Die Simulationsdaten an sich bestehen hierbei aus einer großen Menge nur schwer direkt interpretierbarer Zahlenwerte, die in der Regel als Datei gespeichert werden.

Anschließend ist es die Aufgabe der Strömungsdatenvisualisierung, aus dieser räumlich und zeitlich diskretisierten Datenmenge mit Hilfe von mathematischen Methoden eine interpretierbare Darstellung bzw. die dazugehörige Geometrie, die dann zur Ausgabe an die Graphikkarte gesendet wird, zu berechnen. Die hierbei verwendeten mathematischen Methoden, die zur Erzeugung eines interpretierbaren Bildes aus der zur Verfügung stehenden Strömungsdatenmenge führen, werden an dieser Stelle mit dem Begriff *Visualisierungsmethode* bezeichnet.

**Definition 3.1 (Visualisierungsmethode)** *Mit dem Begriff Visualisierungsmethode werden in dieser Arbeit die mathematischen Methoden und Algorithmen bezeichnet, die aus einer vorgegebenen Strömungsdatenmenge eine Darstellung bzw. die dazugehörige Geometrie generieren. Hierbei ist der verwendete Algorithmus der jeweiligen Visualisierungsmethode meist parametrisiert, d.h. mit Parametern zur Steuerung des errechneten Erscheinungsbildes ausgestattet.*

Diese Visualisierungsmethoden sind meist nicht nur „exklusiv“ auf einen Datensatz anwendbar. Vielmehr können die Daten varieren, jedoch die Darstellung, d.h. Visualisierungsmethode, beibehalten werden. Eine einmal entwickelte Methode der Strömungsdatenvisualisierung kann auf unterschiedliche Datenfelder immer wieder erneut angewendet werden. Auch die gleichzeitige Darstellung mehrerer Methoden auf demselben Datenfeld ist möglich, genau wie die mehrmalige Anwendung derselben Methode mit unterschiedlich justierten Steuerparametern.

Bei den eingesetzten Visualisierungsmethoden muss, wie schon in Kapitel 2 angedeutet, zwischen der rein *technisch-wissenschaftlichen* und der *realitätsnahen* Visualisierung unterschieden werden.

**Definition 3.2 (Technisch-wissenschaftliche Visualisierung)** *Die technisch-wissenschaftliche Visualisierung von Strömungsdaten dient dem Zweck, die visualisierten Strömungsdaten nach technisch-wissenschaftlichen Aspekten zu untersuchen. Hierbei werden die simulierten Daten abstrakt und realitätsverfremdet dargestellt (beispielsweise mit Falschfarben), um die dahinterstehende Information zu verdeutlichen bzw. erkennbar zu machen.*

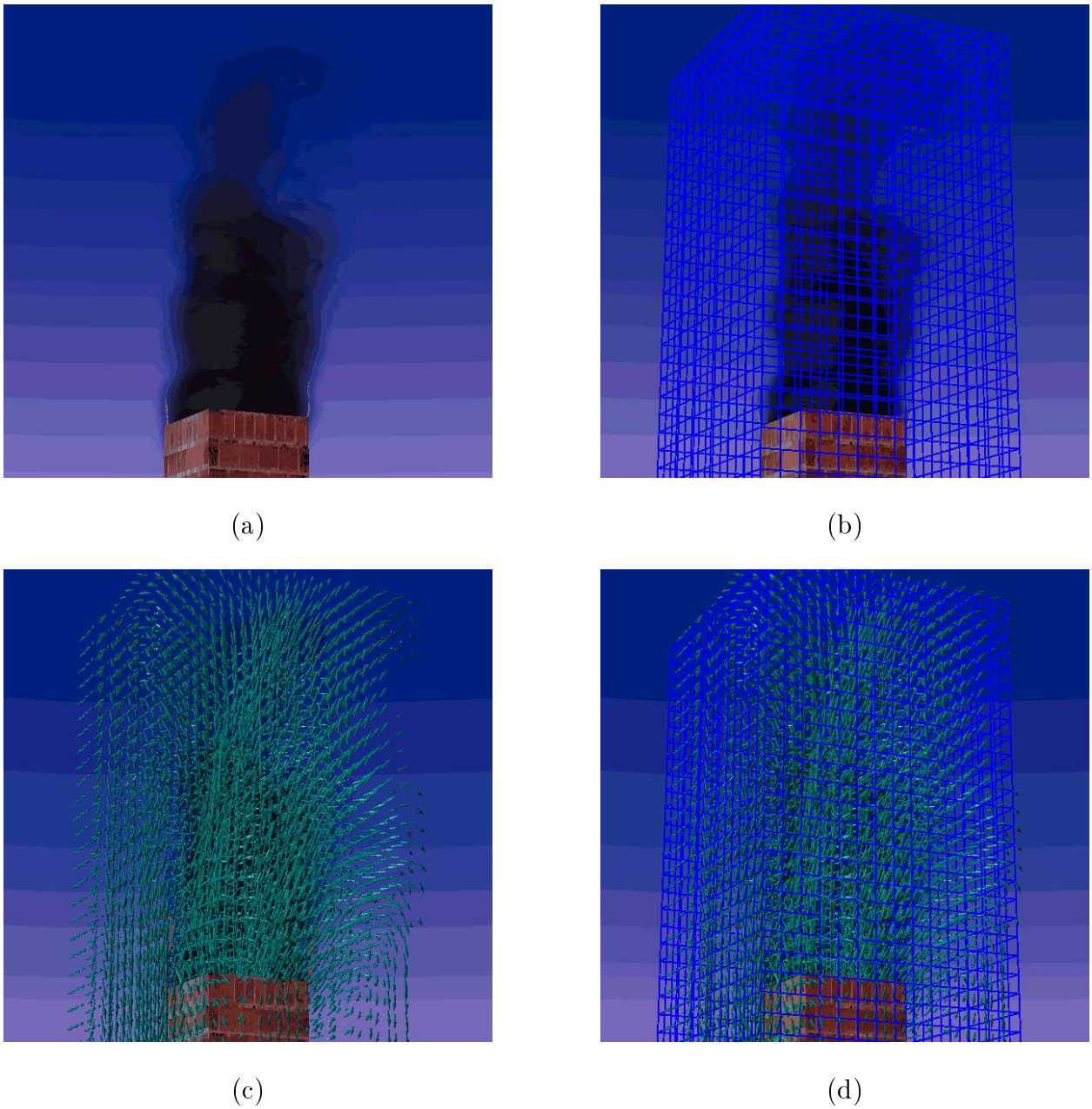
**Definition 3.3 (Realitätsnahe Visualisierung)** *Bei der realitätsnahen Visualisierung von Strömungsdaten steht deren glaubhafter optischer Eindruck im Vergleich zum entsprechenden Erscheinungsbild in der Realität im Vordergrund. Auch wenn die zugrundeliegende Simulation diesselbe ist, wie sie bei der technisch-wissenschaflichen Visualisierung zum Einsatz kommt, so werden die Visualisierungsmethoden, die für die Darstellung der Daten zum Einsatz kommen, vorwiegend darauf abgestimmt, einen möglichst glaubhaften, d.h. „realistisch wirkenden" Eindruck auf den Betrachter zu machen.*

Auch wenn sich diese beide Bereiche leicht voneinander trennen lassen, sind im praktischen Einsatz auch Mischformen möglich, d.h. Darstellungen in denen sowohl technisch-wissenschaftliche, als auch realitätsnahe Visualisierungen gleichzeitig zum Einsatz kommen (siehe Abbildung 3.2 und auch Kapitel 4.1.4). Jedoch ist derzeit keine Software auf dem Markt, die die Mischung beider Visualisierungsarten gleichzeitig unterstützt.

## 3.2 Wissenschaftliche Einordnung

In diesem Abschnitt wird beschrieben, welche verschiedenen Techniken zur Visualisierung von Strömungsdaten bereits exisitieren und in der Praxis zum Einsatz kommen. Diese bereits verfügbaren Lösungen werden kategorisiert und analysiert. Ihre verschiedenen Ansätze und die damit einhergehenden Vor- und Nachteile der dahinterstehenden Konzepte werden beleuchtet. Das alles dient der wissenschaftlichen Einordnung der hier vorgestellten Arbeit in Bezug auf die bereits vorhandenen Techniken und Strategien, bzw. ihrer Abgrenzung von diesen. Im Bereich der Strömungsdatendarstellung besteht eine große Nachfrage nach entsprechenden Visualisierungstechnologien. Allerdings haben die bereits verfügbaren Techniken neben diversen Vorteilen des jeweils verwendeten Ansatzes die verschiedensten Mängel. Die Nachfrage nach einem Konzept, dass die unterschiedlichen Ideen und Konzepte zu einem Gesamtkonzept integriert und vor allen Dingen auch ihre Probleme abdecken kann, für die derzeit noch überhaupt keine brauchbare Lösung existiert, ist entsprechend groß. Im Kapitel 4 wird deshalb aus der eigentlichen Problemstellung und auf Basis der hier vorgestellten Techniken ein entsprechender Anforderungskalatog für die effektive und leistungsfähige Visualisierung von Strömungsdaten zusammengestellt.

Die verfügbaren Werkzeuge zur Strömungsdatenvisualisierung lassen sich in zwei grobe Kategorien zusammenfassen. Ein „integrierter" und ein „separierter" Visualisierungsansatz ist hierbei erkennbar. Beide werden mit ihren unterschiedlichen Vor- und Nachteilen kurz vorgestellt:



(a) (b)

(c) (d)

***Abbildung 3.2:*** *Mischung verschiedener Visualisierungsarten einer Schornsteinsimulation: (a) Realitätsnahe Darstellung der Umgebungsgeometrie und des Rauches. (b) Hinzugeschaltete Gitterdarstellung der Daten. (c) Zusätzliche Vektorendarstellung des zugehörigen Geschwindigkeitsfeldes. (c) Alle Darstellungsarten zusammen.*

**Integrierter Ansatz**

Heutzutage exisitieren auf dem Softwaremarkt eine Reihe weit entwickelter Werkzeuge und Softwarebibliotheken zur Strömungsdatensimulation. Prominente Vertreter dieser Kategorie sind beispielsweise Fluent [FLU], FemLAB [FEM] oder SC/Tetra [SC/]. Auch Programme wie MathLAB [mat05] oder Maple [map05] lassen sich für die Strömungssimulation einsetzen. Innerhalb der Fraunhofer Gesellschaft ist beispielsweise Visicade [VISb] ein entsprechender Kandidat dieser Kategorie.

Alle diese Werkzeuge haben gemeinsam, dass der eigentliche Kern der Anwendung, die Simulation der Strömung zum Schwerpunkt hat. Die Visualisierung ist zwar mit in das Gesamtsystem integriert, oder wird als kleines Zusatzprogramm direkt bei Kauf der entsprechenden Software mitgeliefert, liegt jedoch nicht im direkten Fokus der Entwicklungen des Softwareherstellers. Die so mitgelieferte Visualisierung ist zudem direkt auf das von der Simulationseite bestimmte, für diese optimierte, Datenformat ausgerichtet.

Der eigentliche Schwerpunkt dieser Softwarepakete liegt klar auf der Simulationseite, weniger auf der angeschlossenen Visualisierung. Die Darstellung der Daten wird somit meist entsprechend stiefmütterlich behandelt. Aufgrund der verschiedenen Systeme und deren unterschiedlicher Datenformate, stellen solche Visualisierungsanwendungen in der Regel „Insellösungen“ dar. Sie sind nur speziell auf einen Anwendungsfall - auf eine Simulationsanwendung bzw. deren Datenformat - ausgerichtet. Diese Visualisierungswerkzeuge sind nur bedingt bzw. meist gar nicht in irgend einer Form an eigene Bedürfnisse anpass- oder erweiterbar. Beispielsweise wird das Umgestalten der graphischen Bedienoberfläche, die Integration neuer Visualisierungsmethoden oder auch die Anpassung an einen bestimmten Anwendungsfall nicht unterstützt. Schwierig bis unmöglich gestaltet sich dabei auch die entsprechende Erweiterung des Systems durch eigene Programmierung.

In der Regel ist bei solchen Systemen das Nachladen und realitätsnahe Darstellen der Umgebungsgeometrie der simulierten Strömungsszene genauso wenig möglich, wie die Integration zusätzlicher realitätsnaher Visualisierungsmethoden neben den vorhandenen technisch-wissenschaftlichen.

Oft mangelt es den vorhandenen Darstellungsmethoden selbst an Qualität oder entsprechendem Detailgrad im Vergleich zur technisch möglichen „State of the Art“ Visualisierung im Computergraphikbereich. Da nunmal die Simulationsseite den Themenschwerpunkt dieser Art von Anwendung darstellt, ist die angeschlossene Visualisierung selten auf dem aktuellen Stand der Technik. Schnell veralten die verwendeten Methoden und hinken dem sich rapide entwickelnden Hardwaremarkt im Computergraphik Bereich und den damit verbundenen graphischen Möglichkeiten hinterher. Aktuelle, effizientere Ansätze und zur Verfügung stehende Konzepte aus dem Visualisierungsbereich finden nur sehr langsam ihren Weg in diese Art von Systemen oder sind bei ihrer Intergration bereits veraltet.

Generell haben diese Visualisierungwerkzeuge Probleme mit großen Simulationsdatenmengen umzugehen. Die Interaktivität der Visualisierung ist meist nur bedingt verfügbar und beschränkt sich, falls vorhanden, auf einfaches Navigieren der vorberechneten Szene (drehen, Ausschnitte vergrößern). Der Grund hierfür liegt darin, dass die verwendeten Datenformate für die schnelle Simulationsverarbeitung und ggf. Speicherung der Daten aus der laufenden Simulation heraus ausgelegt sind, nicht jedoch auf ihre effiziente Visualisierung.

**Separierter Ansatz**

Die bekanntesten Vertreter der von der Simulation an sich getrennt entwickelten Strömungsdatenvisualisierung sind Programme wie die Softwarebibliothek VTK [VTK],

AVS [AVS], OpenDX [OPEb], TecPlot [TEC] oder Visualisierungsumgebungen wie Covise [COV]. Innerhalb der Fraunhofer Gesellschaft wird hierfür beispielsweise am ITWM ein Clustersystem zur Visualisierung eingesetzt [PV-].

Generelles Merkmal dieser Visualisierungswerkzeuge ist es, dass sie losgelöst von einer speziellen Simulationssoftware entwickelt werden. Sie unterstützten oftmals eine Vielzahl von Datenformaten. Die eigentliche Visualisierung steht hier im Vordergrund. Die Probleme der Vertreter des „integrierten Ansatzes“ haben Programme dieser Kategorie nicht, dafür aber andere konzeptuelle Schwächen, auf die hier eingegangen wird.

Die Vertreter dieser Anwedungskategorie sind nicht ohne zusätzlichen Konfigurations- und/oder Programmieraufwand verwendbar [VTK]. Vor dem ersten nutzbaren Einsatz des Systems muss zuerst eine technisch versierte Person Hand anlegen, und die zur Verfügung stehenden Einzelteile der Visualisierungsanwendung korrekt initialisieren und zusammenschalten [TEC, COV]. Es kann auch vorkommen, dass beispielsweise zuerst eine Art Scriptsprache für den Datenimport verwendet werden muss, in der der Aufbau der zu lesenenden Daten bzw. die auf ihn anzuwendenden Visualisierungsmethoden samt zugehöriger Parameter definiert werden müssen [OPEb].

Auch in diesem Bereich lässt die zur Verfügung stehende Interaktivität der Visualisierung der untersuchten Daten stark zu wünschen übrig. Bei der Verwendung von großen Datenmengen scheitern auch diese Werkzeuge, da heutzutage noch kein etabliertes Konzept für den Umgang mit großen Strömungsdatenmengen auf dem Softwaremarkt verfügbar ist.

Die qualitativ hochwertige Darstellung der Umgebungsgeometrie der Simulationsszene wird selten unterstützt, genausowenig wie das Zuschalten realitätsnaher Visualisierungsmethoden.

Die Visualisierungsmethoden sind in der Regel an konkrete Systemvorgaben gebunden, somit nicht platformübergreifend universell einsetzbar. Es werden bestimmte Betriebssysteme, bestimmte Szenegraphen und manchmal auch bestimmte Hardwaregegebenheiten für den Betrieb vorausgesetzt. Die zur Verfügung stehenden Visualisierungsmethoden sind nicht in andere, bereits bestehende Systeme und Anwendungen integrierbar. Selten ist die Bedienoberfläche an spezielle Bedürfnisse anpass- oder erweiterbar.

Der wichtigste Vorteil gegenüber dem „integrierten Ansatz“ ist jedoch der, dass die Software, die dieser hier vorgestellten Kategorie angehört, getrennt von der Simulation entwickelt wird. Sie ist nicht an den Fortschritt eines speziellen Simulationswerkzeuges oder dessen Hersteller gebunden und kann somit schneller auf Veränderungen im Graphikbereich und der dazugehörigen Technik reagieren. Da die Graphik den Schwerpunkt und die Grundlage der Visualisierung darstellt, ist dieser Vorteil entscheidend.

Im Anschluß wird an dieser Stelle noch auf einige der oben beschriebenen Werkzeuge und Konzepte zur Strömungsdatenvisualisierung eingegangen und diese genauer vorgestellt. Wie beschrieben, unterscheiden sie sich nicht nur anhand der Qualität und Anzahl der jeweils enthaltenen Visualisierungsmethoden, sondern auch in der Leistungsfähigkeit beim Umgang mit großen Datenmengen, dem Interaktionsgrad, der Erweiterbarkeit z.B. im Bezug auf die Integration in andere Umgebungen und anhand der eingesetzten Konzepte.

### 3.2.1 Covise

Covise [COV] ist eine Visualisierungssoftware für den kolaborativen Einsatz in einem Netzwerk, die diverse VR Geräte wie beispielsweise eine CAVE [CAV] unterstützt. Covise kann unter anderem auch für die Ausgabe von Strömungssimulationsdaten verwendet werden. Entwickelt wurde Covise an der Universität Stuttgart. 1997 führte das Produkt zur Gründung der Firma VirCinity und ist für Lunix / Unix Umgebungen verfügbar. Es zählt zu den Vertretern der in Kapitel 3.2 kategorisierten „von der eigentlichen Simulation separierten", d.h. unabhängigen Visualisierungswerkzeugen.

Die Covise Software besteht aus einzelnen Modulen, die für bestimmte Aufgabenbereiche eingesetzt werden können. So enthält der „Basic" genannte Teil das Kernstück des Visualisierungssystems. Darüber hinaus gibt es dann Module, mit deren Hilfe man Finite Element Daten oder VRML Daten in das System bringen kann. Auch für CFD Daten steht so ein Modul zur Verfügung, mit dessen Hilfe man CFD Daten in das System zur Visualisierung einladen kann. Hierbei wird der Import bestimmter Datenformate (siehe Kapitel 3.3) mit der vom Hersteller „Modul" genannten entsprechenden Softwareerweiterung in das System gebracht.

Möchte man Covise einsetzten, so sind die dazu nötigen Handgriffe vom Prinzip her die folgenden: Zunächst muss der Anwender in einer Art „Schaltplan" die zur Auswertung der Daten benötigten Module von Covise grafisch miteinander verbinden (siehe Abbildung 3.3 links unten). Die einzelnen Module, die hierbei „verschaltet" werden, haben in der Regel mehrere Ein- und Ausgabe-„Slots" (in der Graphik durch rechteckige Kästchen oben und unten an jedem Modul dargestellt). Welcher dieser Slots welche Funktion im Modul erfüllt, ist hierbei aufgrund ihrer Fülle unter Umständen schwierig nachzuvollziehen, jedoch wird hierbei die Typengleichheit geprüft, so das die ein und ausfließenden Parameter zueinander „passen" müssen (z.B. Skalarfelder als Ein- und Ausgabedaten). Jedoch kann es vorkommen, dass ein Modul mehrere Ausgabefelder hat, von denen nur manche besetzt sind. Der Rest der Slots läßt sich unter Umständen auch verbinden, jedoch wird auf dieser Verbindung später keine Information transportiert. Per Rechtsklick auf ein solches Modul erschließen sich dem Anwender zusätzliche Steuerparameter des einzelen Moduls in einem extra Menüfenster, dass diese Parameter enthält. Diese per extra Fenster zugänglichen Parameter lassen sich mit etwas Aufwand auf diverse Bedienelemente in der anschließenden grafischen Ausgabe umleiten.

Die Grundidee von Covise ist die möglichst granulare Zerlegung der einzelnen Schritte, die am Ende zu einer Visualisierungsmethode führen. Diese einzelnen Schritte finden sich im Programm dann alle in einzelnen „Modulen" wieder, die einzeln eingebaut und aufeinander eingestellt werden müssen, um an Ende eine Visualisierung zu erhalten.

Zum Einlesen der Daten wird beispielsweise ein „ReadFluent" Modul, das also Fluent [FLU] Daten einladen kann, in das System gebracht. Anschließend werden an dieses Modul diverse andere Module angeschlossen. Will man beispielsweise eine Vektordarstellung der Geschwindigkeiten auf einer bestimmten Ebene im Raum erreichen, so muss zunächst ein Schnittebenenmodul die entsprechende Ebene aus dem Datenraum „schneiden". Danach wird mit ihm ein weiteres Modul, das „Vektorpfeildarstellungsmodul", verschaltet. Nach wie vor ist stets


***Abbildung 3.3:*** *Covise User Interface: Links unten im Bild der „Schaltplan“, der miteinandern verbundenen einzelnen Module, rechts ein sogenannter „Renderer“, der zur Ausgabe der Daten dient. Oben: Der eigentliche Schwerpunkt von Covise: Die kolaborative Visualisierung.*

darauf zu achten, die richtigen Ein- und Ausgabe-„Slots“ miteinandern zu verbinden und die Parameter der einzelnen Module korrekt über diverse Kontextmenüs einzustellen. Möchte man die Vektoren nun anhand eines bestimmten Kriteriums einfärben, so ist hierfür auch ein weiteres entsprechendes „Einfärbe“Modul von Nöten, dass selbst wiederum einen Eingabeparameter per Slot von einem anderen Modul benötigt, das wiederum festlegt, nach welchem Kriterium diese Einfärbung geschehen soll. Nachdem eine Reihe solcher Module miteinander verschaltet wurden, kommt ein „Collector“ genanntes Modul zum Einsatz. Dieses Modul bereitet die berechneten Daten zur Darstellung vor. Anschließend wird ein „Renderer“ Modul mit dem ganzen Schaltplan verbunden. Seine Aufgabe ist es, diese Daten auf einem festgelegten Gerät auszugeben. Das „Cover“ genannte Renderer Modul unterstützt beispielsweise die Ausgabe auf dem Monitor oder in der Cave [CAV].

Die Konstruktion des für die Darstellung benötigten Schaltplanes kann sich für den nicht versierten Anwender als recht schwierig bis nicht durchführbar gestalten. Aufgrund der Vielfalt der vorhandenen Slots, deren Benennung auch nicht immer direkt aufschlußreich sein

muss und der Möglichkeit, auch „leere“ Slots miteinander verbinden zu können, kann es einige Zeit in Anspruch nehmen, bis die gewünschte Visualisierung ihren Weg auf den Bildschirm des Anwenders findet, da selbst für eine einfache Visualisierung, wie oben im Beispiel beschrieben, bereits eine Fülle von unterschiedlichen Modulen miteinander verschaltet und ihre entsprechenden Parameter an unterschiedlichen Stellen eingesteuert werden müssen. Für einen schnellen Einsatz, d.h. das sofortige Untersuchen der Daten mit einer bestimmten Visualisierungsmethode, sowie das interaktive „Spielen“ mit den entsprechenden Visualisierungsparametern, ist Covise deswegen nur bedingt geeignet. Ein Grundproblem hierbei ist, dass immer nur ein einziger Zeitschritt der untersuchten Simulationsdaten den Modulschaltplan durchläuft. Anschließend wird dieser Zeitschritt visualisiert. Erst sobald ein Zeitschritt erfolgreich durch die Modulschaltung geschleust wurde, wird der nächste Zeitschritt durch den Schaltplan geschickt, um am Ende der Kette den vorangegangen in der Visualisierung abzulösen. Je nach Menge der Daten erreicht dieses Konzept aufgrund der Schnittstellenanzahl zwischen den vielen einzelnen Modulen, Speicherzugriffszeiten, und -größen schnell sein Limit.

Die konstruierten „Schaltpläne“ lassen sich zwar abspeichern und dadurch wiederverwenden, jedoch müssen sie, falls andere Daten untersucht werden sollen, dennoch angepaßt werden, da die interessanten Datenfelder z.B. dann auf einem anderen Slot des Import Modules liegen. Ferner müssen die vielen verschiedenen Parameter der einzelnen Module an den neuen Datensatz angepaßt werden.

Die gleichzeitige Anwendung von mehreren Visualisierungsmethoden auf demselben Datensatz ist möglich. Jedoch steigt mit jeder neuen Methode die Modulanzahl und damit der Aufwand, den entsprechenden Schaltplan zu konstruieren und zu pflegen. Aufgrund der Fülle der entstehenden Verbindungen wird er schnell unübersichtlich. Die Integration einer neuen Visualisierungsmethode in eine bereits aktive Szene, erfordert eine entsprechend aufwendige und damit zeitraubende Erweiterung des Schaltplanes.

Die Aktivierung realitätsnaher Visualisierungsmethoden oder auch das Mischen dieser mit technisch-wissenschaftlichen Methoden ist in diesem System nicht möglich.

Für eine Echtzeitvisualisierung der eingehenden Strömungsdaten ist Covise nur bedingt geeignet, da man - wie beschrieben - zunächst erst den Schaltplan festlegen und dann die verwendeten Module bzw. deren Parameter justieren muss. Anschließend muss der Datenfluß durch die einzelnen Module abgewartet werden, bis schließlich ein Bild vom ans Ende des Schaltplans integrierten Renderer Modul berechnet wird. Von Modul zu Modul werden die veränderten Daten über abstrakte Schnittstellen weitergegeben. Hierbei werden die Daten vor und nach jedem Modul in eine „Datenliste“ gepackt, aus der sie anschließend wieder ausgelesen werden müssen. Mit dem Steigen der Anzahl der vewendeten Module steigt auch die Anzahl der „Ein und Auspack“ Vorgänge der Daten bzw. wird hierfür entsprechend mehr Rechenzeit verbraucht.

Arbeiten mit großen Strömungs-Datenmengen wird zwar vom System unterstützt, da sie von ihm rein technisch gesehen „adressierbar“ sind, jedoch ist es aufgrund fehlender Konzepte zum Umgang mit großen Datenmengen nur bedingt in der Praxis möglich. Dadurch, dass jedes Modul seinen eigenen Ein- und Ausgabeslot bzw. die dazugehörigen Speicherlisten für

die Daten verwendet, steigt auch der entsprechende Speicheraufwand mit jedem neu in den Schaltplan integrierten Modul.

Das Steuern des Teilraumes, auf dem eine Visualisierungsmethode aktiv sein soll, ist nur bedingt möglich. Visualisierungsmethoden in Covise arbeiten immer auf dem gesamten Datenvolumen. Über bestimmte Schnittelemente (z.B. 2D Schnittebenen) lassen sich zwar aus diesem Volumen (erst im Verlauf der Berechnungen) entsprechende Teile herauslösen. Jedoch werden sie zuvor komplett in den Speicher geladen. Elemente, die 3D Teilvolumina aus dem Datenraum herausschneiden stehen nicht zur Verfügung, können aber mit etwas Aufwand selbst implementiert werden. Jedoch würde sich eine entsprechende Positionierung und interaktive Steuerung dieser Teile und deren Integration in den „Schaltplan“ als entsprechend schwierig bis unmöglich gestalten.

### 3.2.2 FEMLAB und Fluent

FEMLAB [FEM] von der Firma Comsol und Fluent [FLU] von der Firma Fluent Inc. sind beide Simulationssoftware Pakete für die Modellierung von physikalischen Prozessen, die sich mit partiellen Differentialgleichungen beschreiben lassen. Neben vielen physikalischen Problemstellungen, lassen sich mit ihnen auch CFD Probleme abbilden. Zur Simulation verschiedener physikaler Situationen stehen je nach Anwendungsgebiet unterschiedliche Erweiterungspakete der Software zum Nachkauf zur Verfügung. Die Visualisierungskomponente bleibt hierbei jedoch dieselbe. Sie gehören zu den Vertretern der hier als „intern“ bezeichnenten Kategorie (siehe Kapitel 3.2).

Aufgrund ihrer Natur, vom Kern her eine Simulationssoftware zu sein, unterstützen die Systeme nur bedingt den Datenimport aus Fremdformaten. Als reine Ausgabe- bzw. Visualisierungswerkzeuge sind sie deswegen nur bedingt einsetzbar. Die zu visualisierenden Daten werden in der Regel vom verwendeten System anhand der Durchführung einer entsprechenden Simulation zuvor selbst generiert. Anschließend kommt die mit der Simulationssoftware mitgelieferte Visualisierungssoftware für die Darstellung der Daten zum Einsatz. Diese arbeitet dann gewissermaßen als Postprozess. Damit die Daten dargestellt werden können, müssen diese also bereits vorberechnet vorliegen. Sie werden wie beschrieben zuvor von der zugehörigen Simulationsanwendung berechnet und abgespeichert. Anschließend werden die Daten in den Visualisierer geladen und definierte Visualisierungsmethoden mit fest eingestellten Parametern darauf angewendet. Erst dann findet die Berechnung der Darstellung statt. Nachdem diese Berechnung beendet ist, läßt sich die fertige 3D Szene betrachten und ggf. rotieren und vergrößern. Möchte man die Visualisierungsmethoden ändern, so ist die erneute Berechnung der gesamten Darstellung nötig. Die eigentliche Visualisierung ist nicht interaktiv steuerbar. Man kann nicht zur Laufzeit der Visualisierung die Parameter der einzelnen Visualisierungsmethoden steuern, die Auswirkungen nicht direkt betrachten.

Die Techniken und der Aufbau der mit der Simulationssoftware mitgelieferten Visualisierungswerkzeuge sind zwar nicht öffentlich zugänglich, jedoch läßt sich sagen, dass auch sie diverse ansatzbedingte Eigenarten mit sich bringen. Neben der oben beschriebenen nur bedingten Anwendbarkeit der Visualisierung auf nicht herstellerspezifische Datenformate, sind

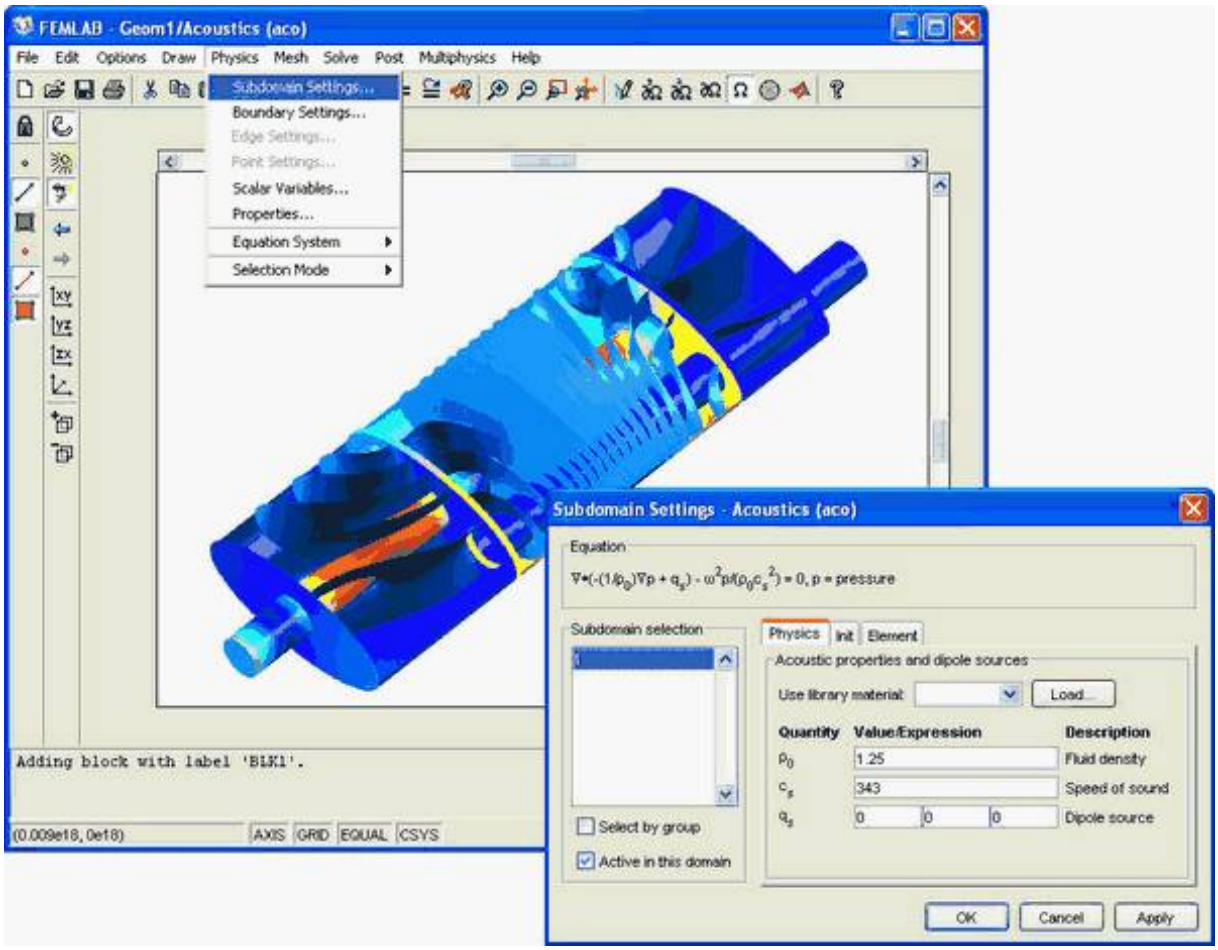



***Abbildung 3.4:*** *FEMLAB Visualisierung: Die untersuchten Daten werden zunächst anhand der Lösung einer partiellen Differentialgleichung vom System errechnet. Anschließend kommt die mitgelieferte Visualisierungssoftware zum Einsatz.*

die Visualisierungswerkzeuge nicht vom Endanwender selbst erweiterbar. Es lassen sich keine neuen Visualisierungsmethoden oder Techniken von ihm nachrüsten. Genausowenig läßt sich die Programmoberfläche bzw. das Benutzerinterface an bestimmte Nutzungsbedingungen anpassen. Die Visualisierungswerkzeuge sind an feste Plattformen gebunden. FEMLAB läuft nur unter Windows [Micb], Fluent steht für Unix/Linux und Windows zur Verfügung.

Konzepte zur Unterstützung großer Datenmengen für die Visualisierung werden nicht angeboten. Je größer die darzustellende Datenmenge, um so langsamer die Berechnungen. Wird die Datenmenge so groß, dass die für die Visualisierung berechnete Szene nicht komplett in den Speicher passt, wird die Darstellung entsprechend ausgebremst bis unmöglich. Die Visualisierungsmethoden lassen sich nur bedingt auf bestimmte räumliche Gebiete einschränken. Zur Berechnung der Darstellung wird die komplette Datenmasse bearbeitet, auch wenn nur Teile von ihr visualisiert werden. Eine Parallelisierung der Berechnungen für die Visualisierung ist nicht vorhanden.

Über die technisch-wissenschaftliche Darstellung hinausgehende realitätsnahe Visualisierungs-

methoden werden nicht unterstützt. Auch die Darstellung der zugehörigen Umgebungsgeometrie der simulierten Szene ist nicht bzw. nur eingeschränkt/indirekt möglich.

### 3.2.3 VTK

VTK [VTK] das „Visualization ToolKit“ von Kitware Inc. ist eine Open Source Software für 3D Computergraphik, Bildverarbeitung und Visualisierung. Es steht als Programmier-Bibliothek in der Sprache C++ im Quellcode zur Verfügung. Lauffähig ist es auf nahezu allen Plattformen und Betriebssystemen (Linux/Unix, Windows und Mac OS). Es ist ein Vertreter der „separaten“ Visualisierungswerkzeuge (siehe Kapitel 3.2). VTK selbst ist eine Softwarebibliothek, die im Quellcode verfügbar ist. D.h. es ist keine Visualisierungsanwendung an sich, damit für den Endanwender weniger geeignet, sondern eher eine Programmierhilfe für Entwickler. Eine Reihe von Beispielanwendungen, die auf VTK basieren sind als Anschauungsbeispiele für den Programmiereinsteiger verfügbar.

Trotz seiner Vielfältigkeit ist VTK speziell unter dem Vorzeichen der Analyse großer Strömungsdatenmengen nur bedingt zum Einsatz geeignet. Es bietet zwar die Möglichkeit eine Vielzahl von unterschiedlichen Datenformaten einladen zu können, jedoch benutzt VTK zur internen Weiterverarbeitung der importierten Daten ein eigenes proprietäres Format. Dieses Format ist, genau wie bei allen anderen heutzutage zur Verfügung stehenden Visualisierungswerkzeugen, nur bedingt für die Analyse großer Datenmengen geeignet. Auch hier wird die gesamte Strömungsdatenmenge im Speicher gehalten, selbst wenn die aktiven Visualisierungsmethoden nur auf einem Teil der Daten arbeiten. Die mitgelieferten Visualisierungsmethodencodes sind nicht parallelisiert. Jedoch ist es aufgrund der Verfügbarkeit des Quellcodes möglich, eigene Methoden zu implementieren.

Die Integration neuer Visualisierungsmethoden in die Bibliothek ist möglich, da sie im Quellcode zugänglich ist. Von Hause aus werden keine realitätsnahen Visualisierungsmethoden unterstützt, jedoch die wichtigsten technisch-wissenschaftlichen Darstellungen sind bereits in der Bibliothek integriert.

Wie beschrieben, besteht VTK nicht aus einer zentralen Anwendung, sondern vielmehr aus einer Softwarebibliothek, für deren Einsatz Programmierarbeit von Nöten ist. Zwar ist VTK dadurch vielseitig einsetzbar, jedoch ist für jeden neuen Anwendungsfall eine erneute Entwicklung der entsprechenden angepaßten Anwendung von Nöten. VTK selbst kann nicht direkt als Visualisierungsumgebung betrachtet werden, es stellt vielmehr die Mittel zur Verfügung, die zur Implementation einer solchen von Nöten sind. Die Fülle des Codes (derzeit über 110.000 Zeilen ausführbarer Code) gestaltet eine schnelle Einarbeitung in die Bibliothek teilweise als schwierig. Jedoch steht umfangreiches Dokumentationsmaterial - auch in Buchform - zur Verfügung.

### 3.2.4 OpenDX

OpenDX [OPEb], ein Open Source Projekt, das auf dem IBM Visualization Data Explorer [IBM] basiert, ist eine Softwareanwendung, für die Visualisierung von wissenschaftlichen,

konstruktions und analytischen Datenmengen. Es kann demzufolge auch zur Darstellung von Strömungsdatenmengen verwendet werden. Unterstützt werden alle Plattformen und Betriebssysteme. Nach der Kategorisierung, wie sie in Kapitel 3.2 eingeführt wurde, zählt OpenDX zu der „separaten“ Version.

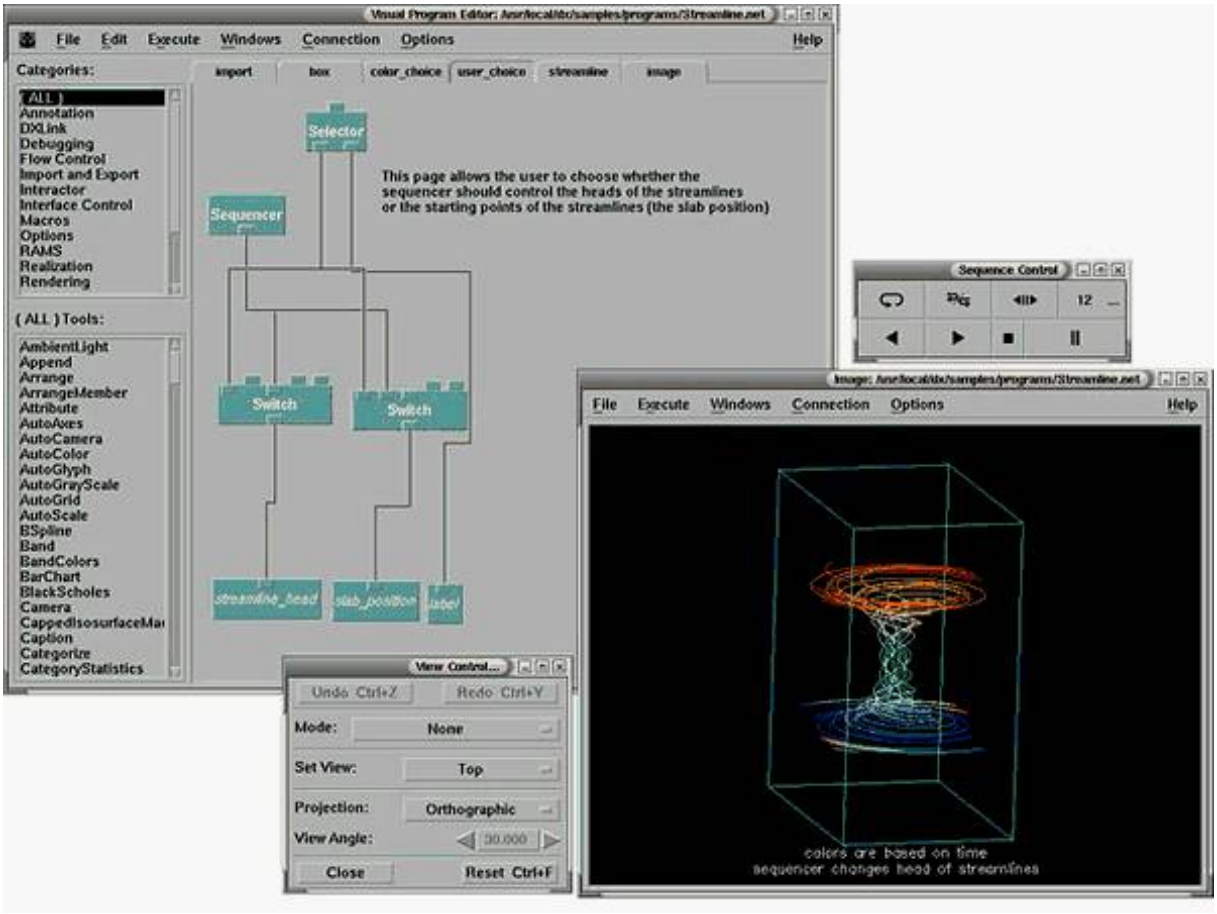



***Abbildung 3.5:*** *OpenDX: Die Programmoberfläche besteht aus einem „Schaltplan“ Bereich, in dem die Visualisierungsmethoden mit den Daten „verschaltet“ werden. Anschließend werden die Daten in einem separaten Renderer dargestellt.*

Hier wird, ähnlich wie bei der Verwendung von Covise (vgl. Kapitel 3.2.1), zunächst ein Schaltplan angelegt, der zur Generierung der für die Visualisierung bestimmten Daten durchlaufen wird. Die Erstellung und Pflege dieses Schaltplanes gestaltet sich auch ähnlich aufwendig. Zur besseren Übersichtlichkeit werden hier die für die einzelnen Visualisierungsmethoden benötigten Module auf „Karteikärtchen“, d.h. einzelnen Unterseiten des Schaltplanes, zusammengefasst.

Auch hier kann man in der an die vorgeschaltete Berechnung anschließenden Visualisierung grundsätzlich nur die Szene rotieren, bewegen und vergrößern. So ist beispielsweise eine gewünschte Einschränkung der verschalteten Visualisierungsmethoden auf bestimmte Datenbereiche entsprechend schwierig zu realisieren.

Große Datenmengen bereiten OpenDX, genau wie allen anderen Visualisierungswerkzeugen,

Probleme. Parallelisierung wird in sofern unterstützt, dass einzelne Visualisierungsmethoden auf getrennte Prozessoren oder Rechner ausgelagert werden können. Das entspricht einer der drei Parallelisierungsebenen (die zwischen den Visualisierungsmethoden), die in Kapitel 5.5.1 vorgestellt werden.

Realitätsnahe Visualisierungsmethoden werden, genau wie eine Darstellung der Umgebungsgeometrie, nicht unterstützt.

### 3.2.5 TecPlot

TecPlot [TEC] von der Firma TecPlot Inc. ist eine Visualisierungssoftware für Simulations- und Messdaten. Die spezielle Erweiterung „CFD Analyzer“ eignet sich zum Darstellen von Strömungsdaten. Unabhängig von einer bestimmten Simulationssoftware, gehört auch TecPlot der Kategorie der „separierten“ Visualisierungswerkzeuge aus Kapitel 3.2 an. Es ist für viele Plattformen und Betriebssysteme verfügbar.

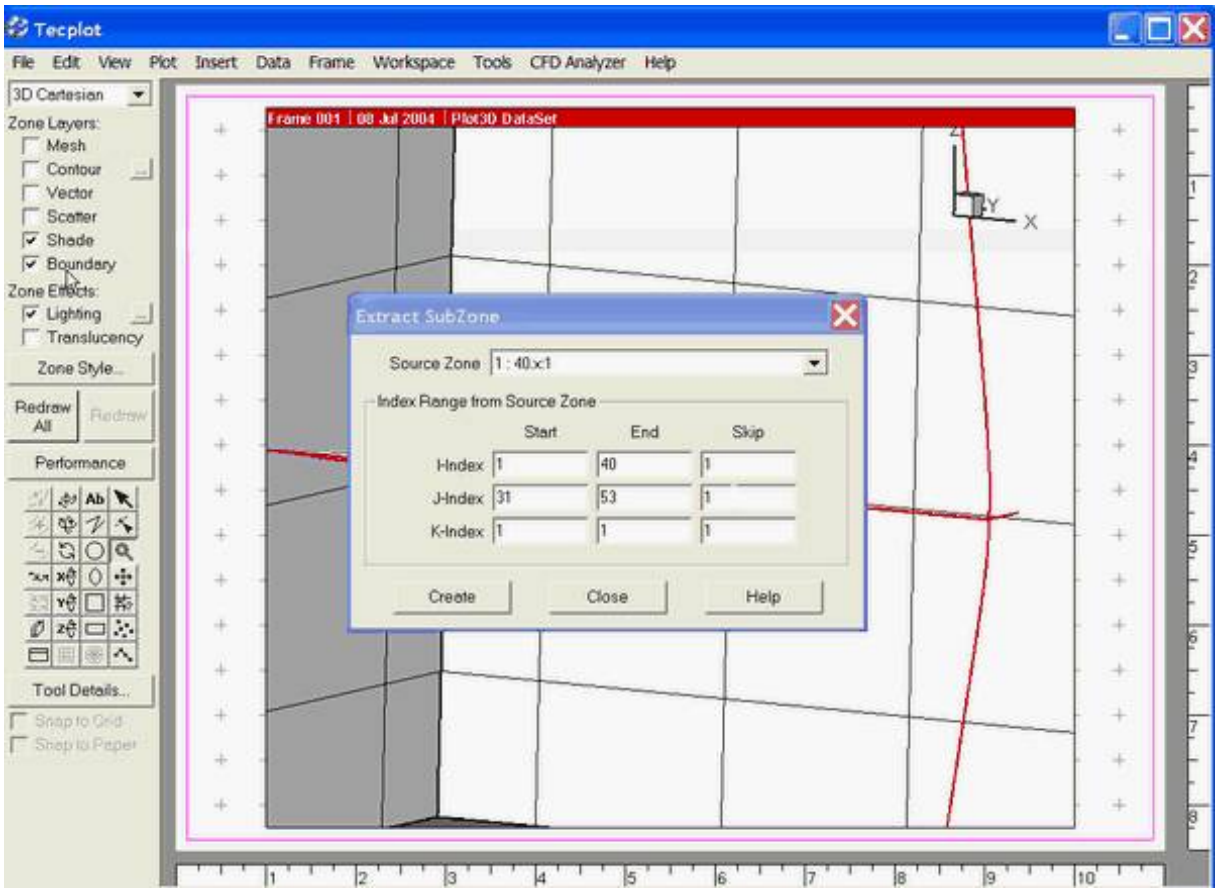



***Abbildung 3.6:*** *TecPlot im Einsatz: Hier besteht die Möglichkeit die zu visualisierenden Daten vor Beginn der eigentlichen Visualisierung in einem Pre-Prozess zu „beschneiden“.*

Bei der Verwendung von TecPlot hat der Benutzer die Möglichkeit, die Daten vor dem eigentlichen Start der Visualisierungsberechnungen zu „beschneiden“ (siehe Abbildung 3.6).

Dies geschieht zwar in einem Pre-Prozess, jedoch hat der Hersteller die Notwendigkeit dieses Vorgehens gerade bei großen Datenmengen erkannt und auf diese Weise umgesetzt. Allerdings wird dieses Beschneiden in erster Linie zur Verringerung des Datenaufkommens verwendet und nicht etwa zum einschränken der Datenmasse an die tatsächlich von der Visualisierung benötigten Datenteile, falls die Visualisierung beispielsweise nur in einem Teilraum aktiv sein soll. Soll der so „beschnittene“ Teilraum geändert werden, muss der Benutzer erneut zu diesem Pre-Prozess zurückkehren und ihn von Hand neu durchführen.

TecPlot liegt zwar nicht im Quellcode vor, verfügt jedoch über ein „Add-on Developers Kit (ADK)“, mit dessen Hilfe sich unter den Programmiersprachen C/C++ und Fortran eigene Erweiterungen in das System zum Teil einpflegen lassen. Eine Erweiterung des Systems ist dadurch in gewissem Umfang möglich, jedoch ist beispielsweise eine Ingegration der von TecPlot bereitgestellten Visualisierungsumgebung in ein anderes System und/oder die Anpassung der Programmoberfläche an bestimmte Bedürfnisse nicht möglich.

Vom Hersteller werden keine Aussagen zur Parallelisierung des System und über die Unterstützung großer Datenmengen, beispielsweise durch intelligente Ein- und Auslagerungsstrategien unbenötigter Datenraumteile, gemacht. Eine Integration eigener, beispielsweise realitätsnaher Visualisierungsmethoden ist, genau wie die Darstellung der eigentlichen Umgebungsgeometrie der Simulationsszene, nicht möglich.

### 3.2.6 Weitere Softwarewerkzeuge und Ansätze

Neben den hier vorgestellten Konzepten zur Darstellung von Strömungsdaten existieren noch eine Reihe weiterer Softwarewerkzeuge und Ansätze. Wie schon in Kapitel 3.2 beschrieben, lassen sie sich in „integrierte“ und „separierte“ Ansätze gliedern:

**Integriert**

Visicade [VISb] ist ein Projekt des Fraunhofer IGD zur Schaffung einer Simulationsumgebung für die nahtlose Integration von CAD/CAE in VR. CAD Modelle können zum Start der Simulation geladen werden, während die parallelisierte Visualisierung die entsprechend erzeugten Daten anhand einer festen Auswahl an Methoden darstellt. SC/Tetra [SC/] ist eine CFD Simulationsumgebung mit integrierter Visualisierungsmöglichkeit. Diese wird als sogenannter „Postprozessor“ mitgliefert und bietet eine feste Auswahl an zur Verfügung stehenden Visualisierungswerkzeugen. PowerFlow [POW] und das zugehörige PowerViz bieten ein eng verbundenes System zur Strömungssimulation und -visualisierung. Auch hier steht der direkte Aufbau der Visualisierung auf ein bestimmtes Visualisierungssystem und dessen Datenformat im Vordergrund der Entwicklung. Verschiedene Visualisierungsmethoden werden vom fertigen Komplettsystem unter Unix und Windows bereitgestellt.

**Separiert**

AVS [AVS] Express ist eine Visualisierungsumgebung ähnlich wie OpenDX, da sie als fertige Lösung erhältlich ist, jedoch als Developer Version auch mit eigenen Erweiterungen ausgestattet werden kann. Der OpenSG Szenegraph [OPEc] ist zwar kein

Visualisierungssystem als solches, aber ein nützliches Werkzeug zur Visualisierung, das von der Fraunhofer Gesellschaft entwickelt wird. In seiner neuesten Version unterstützt er die parallele Ausgabe der fertigen im Szenegraph abgespeicherten Visualisierungsdaten per Cluster [HEY, Rot03]. Eine andere Lösung wird vom Fraunhofer Insitut für Techno- und Wirtschaftsmathematik mit seinem CC HPC (Competence Center High Performance Computing und Visualisierung) bzw. dem dort befindlichen Visualisierungscluster und der dort unter Linux entwickelten Visualisierungssoftware PV-4D [PV-] bereitgestellt. Hier ist die der Visualisierung vorgeschaltete Berechnung parallelisiert, und wird zusammen mit entsprechenden Simulationsprogrammen weiterentwickelt, wobei das System speziell auf das Volumenrendering ausgelegt ist (vgl. Kapitel 7.1.5 und 7.1.6.1).

Darüber hinaus beschäftigen sich noch eine Reihe weiterer wissenschaftlicher Arbeiten mit dem Thema der parallelen Datenvisualisierung [psv05, gig97, SRBE99, FL99], insbesondere der Strömungsdatenvisualisierung. Insgesamt ist die parallelisierte Visualisierung von Strömungsdaten ein sehr großes Forschungsgebiet, auf dem eine Vielzahl unterschiedlicher, sehr spezifischer Lösungen enwickelt wurden und werden. Hier gilt es, immer genau den Anwendungsfall und das verfolgte Ziel, für das die jeweilige Visualisierung optimiert wird, zu unterscheiden. Die Integration der zur Verfügung stehenden Techniken in ein Gesamtsystem und die Entwicklung neuer Techniken für Probleme, für die es bisher noch keine sinnvolle Lösung gibt, ist der Schwerpunkt dieser Arbeit.

## 3.3 Datenformat

Wie bereits in Kapitel 3.1 angedeutet, ist es heutzutage die gängige Vorgehensweise, dass die Simulation ihre errechneten Strömungsdaten bzw. Felder zu festgelegten Simulationszeitpunkten abspeichert. Hierbei wird der simulierte Raum bzw. das zugehörige Volumen in Gitter unterteilt. Die ermittelten Werte der Simulation werden auf diese Gitter diskretisiert und abgespeichert. Auf den Elementen des Gitters sitzen die Datenwerte 3.7. Hierbei unterscheiden sich die Verfahren darin, ob die berechneten Werte auf den Gitter-Eckpunkten, den Kanten oder auch mittem im Volumen der einzelnen Zelle sitzen. Auf Grundlage dieser Gitter rechnet die Simulation die zu untersuchende Szene durch.

Die bei der Strömungssimulation berechneten Felder können sehr unterschiedlich sein. Das Spektrum reicht von Skalarfeldern, die beispielsweise Temperatur-, Druck- oder Dichteinformationen des strömenden Materials enthalten, über Vektorfelder (z.B. die Strömungsrichtung und Geschwindigkeit) bis hin zu abgeleiteten Größen in Tensor oder anderen mehrdimensionalen Feldern.

Ein von der Simulation durchgerechnetes Szenario, in dem mehrere dieser Felder enthalten sein können, wird in mehreren Dateien (z.B. eine pro Simulationszeitschritt) abgespeichert. Ein Paket solcher Dateien, die bzgl. eines simulierten Szenarios zusammengehören, werden auch als *Datensatz* bezeichnet.

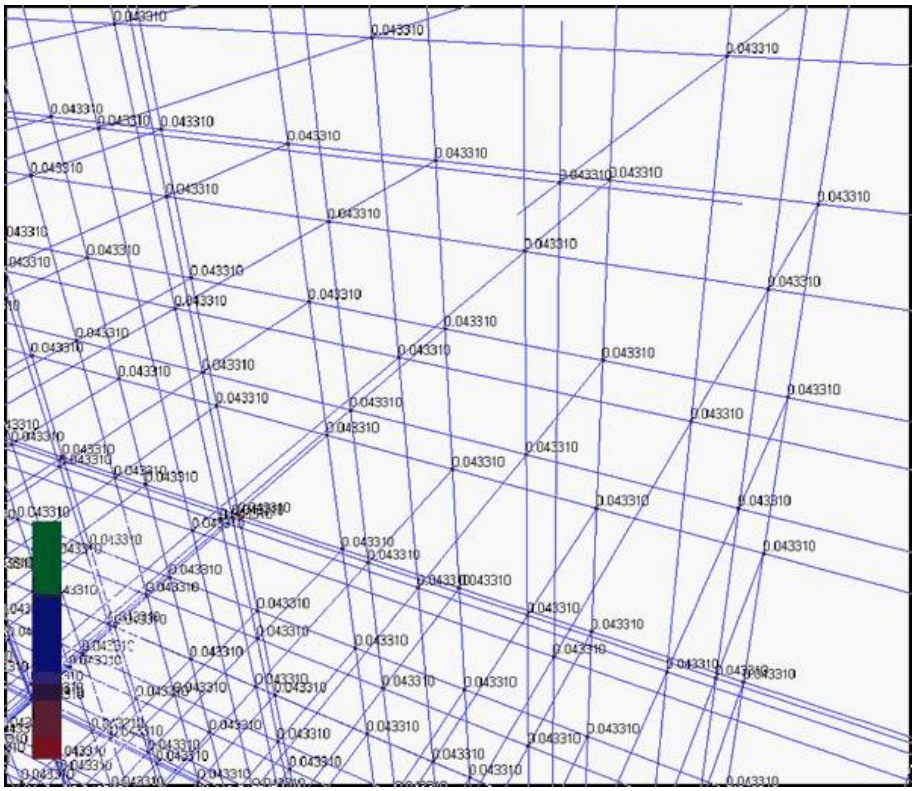



***Abbildung 3.7:*** *Datengitter: Auf den Elementen des diskretisierten Volumens (hier Punkte) sitzen die von der Simulation errechneten Werte.*

Wie schon in Kapitel 1 beschrieben, erzeugen die Simulationsprogramme verfahrensbedingt große Datenmengen. Da sie das simulierte Volumen sowohl räumlich als auch zeitlich diskretisieren, steigt die Datenmenge automatisch, je feiner die Auflösung der Daten gewählt wird. Im Zuge der fortschreitenden Leistungssteigerung der zur Verfügung stehenden Hardware und der Weiterentwicklung der Simulationswerkzeuge, steigt diese Datenmenge immer weiter an. War es früher nur möglich kleine Teilbereiche von Strömungen zu simulieren, so werden die Aufgaben immer komplexer, die Anwendungsgebiete immer vielseitiger (vgl. Kapitel 2).

Die Datenformate, die von der Simulation verwendet werden, sind auf die schnelle und effiziente Speicherung ausgelegt. So wird die Simulation zwischen zwei Berechnungsschritten durch die Datenspeicherung nur kurz ausgebremst, um sich dann wieder völlig der Berechnung widmen zu können. Das hat Auswirkungen auf die anschließende Visualisierung, wie in Kapitel 4.2.9 und 5.6 beschrieben.

Im Laufe der Zeit haben sich die verschiedensten Datenformate etabliert. Auch wenn es kein echtes Standardformat zur Speicherung von Simulationsdaten gibt, so existieren mittlerweile eine Reihe von definierten Speicherformaten, die sich im Bereich der Strömungssimulationssoftware durchgesetzt haben. Zu nennen sind hier beispielsweise Formate wie TecPlot [TEC] oder das weit verbreitete CGNS [cgn05] Format, dass 1994 in einer Zusammenarbeit zwischen *Boing* und der *NASA* entstand. Seit 1999 ist es eine Art „quasi offener Standard“.

Auch CFX [cfx05], das Fluent [FLU] Format oder Plot3D [plo05] sind, neben vielen weiteren Vertretern, bekannte CFD Datenformate. Viele Simulationswerkzeuge verwenden ein eigenes Format zur Datenkonservierung. Manche sind jedoch imstande ihre Daten auch in einigen anderen Formaten zu speichern. In der Regel unterstützen die eigenständig entwickelten („separierten“) Visualisierungswerkzeuge für Strömungsdaten die Fähigkeit, mehrere verschiedene dieser Formate einlesen zu können (vgl. Kapitel 3.2).

### 3.3.1 Gittertypen

Genau so unterschiedlich wie die Datenformate sind auch die verwendeten Gittertypen, die in diesen Formaten abgespeichert werden. Das Spektrum hierbei ist sehr vielseitig und kann sich jeweils pro Anwendungsfall unterscheiden. Auch aus diesem Grund sind einige der verbereiteteren Datenformate explizit sogar darauf ausgerichtet, verschiedene Gittertypen zu unterstützen [mpc05].

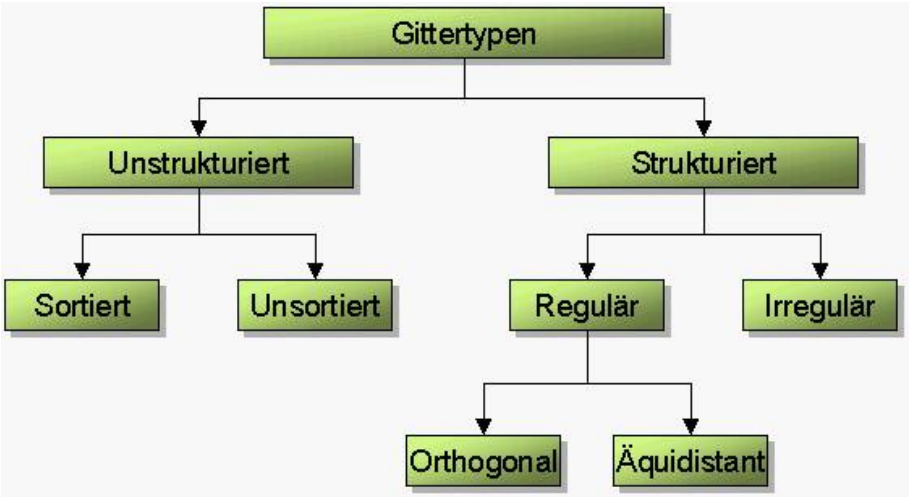



***Abbildung 3.8:*** *Verschiedene Gittertypen.*

Die einzelnen Gittertypen lassen einfach anhand ihrer Strukturierung und/oder den verwendeten Zelltypen unterschieden (Abbildung 3.8). Als Zelltypen kommen die verschiedensten Elemente in Betracht. Die Möglichkeiten gehen über Punkte, Quader, Tedraeder, Octaeder bis hin zu beliebig geformten Elementen. Darauf aufsetzende Interpolationsmethoden, die zum Auslesen der Daten benötigt werden (siehe Kapitel 4.2.9) sind entsprechend aufwendig. Der Prozess, den die Simulation zur Auswertung der Daten eines Gitters durchlaufen muss, ist hierbei immer derselbe und ein zweistufiger Prozess (Abbildung 3.9). Zuerst müssen die relevanten Gitterzellen für die Auswertung bestimmt werden. Anschließend wird anhand dieser Zellen der auszulesende Raum-/Zeitkoordinatenwert ermittelt. In der Regel geschieht das über Interpolation aus den Nachbarpunkten/Zellen/Elementen.

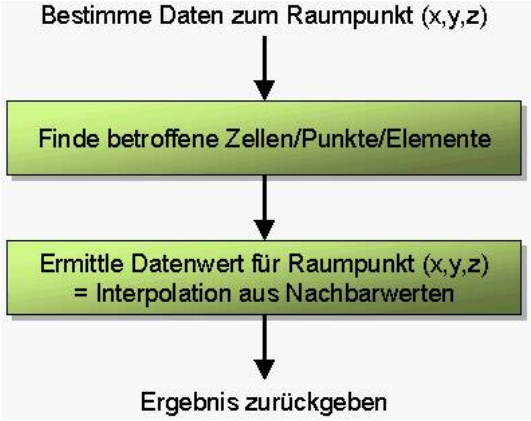



***Abbildung 3.9:*** *Die beiden Schritte, die beim Auslesen eines Gitters durchgeführt werden.*

**Unstrukturierte Gitter**

Das typische Identifizierungsmerkmal für unstrukturierte Gitter ist, dass die verwendeten Zellen nicht räumlich sortiert vorliegen. Selbst wenn die verwendeten Zelltypen alle gleich sind, so bedeutet das Auslesen eines unstrukturierten Gitters für die Visualisierung durch die notwendige und zeitintensive Zellsuche immer erheblichen Aufwand. Nachbarschaftsbeziehungen zwischen einzelnen Zellen müssen vor den Interpolationsvorgängen immer erst zeitaufwendig ermittelt werden. Die Zellen müssen sortiert und durchsucht werden, da diese Informationen dem eigentlichen Datenformat fehlen. Liegen die Gitter zusätzlich noch unsortiert vor, so wird der Aufwand um so größer, die relevanten Zellen um Auslesen der betreffenden Informationen zu ermitteln. In der Regel ist ein effizientes Arbeiten mit unstrukturierten Gittern aus Sicht der Strömungsvisualisierung nur eingeschränkt bis gar nicht möglich. Die zugehörigen Visualisierungen sind langsam und verlieren viel Zeit für die Analyse und das Durchsuchen der dargestellten Gitter.

**Strukturierte Gitter**

Strukturierte Gitter unterscheiden sich in erster Linie in der Anordnung der Zellen gegenüber ihrer unstrukturierten Variate. Hier stehen die abgespeicherten Zellen in einem räumlichen Zusammenhang. Nachbarschaftsbeziehungen sind entweder bereits im Format selbst enthalten oder direkt aus der Anordnung der Zellen ableitbar. Sind die Gitter bzw. Zellen gar regulär, rechtwinklig (orthogonal) und gleich groß (äquidistant), so wird das Bestimmen der betroffenen Zelle(n) beim Auslesen von Datenwerten um so leichter, der Vorgang kann um so schneller ablaufen. Aus Sicht der Visualisierung ist diese Art von Gitter das, was am schnellsten zu verarbeiten ist.

Bei der eigentlichen Speicherung können die Gitter noch in verschiedene Untergitter, auch *Blöcke*, *Subgitter* oder Segmente genannt, unterteilt werden. Dieses Vorgehen dient meist der einfacheren Orientierung im Datensatz oder beispielsweise, wenn bestimmte Bereiche des Gitters mit anderen aktiven Simulationsanwendungen zur Laufzeit Informationen ausstauschen müssen (*Gekoppelte Simulationen*).

## 3.4 Parallele Datenverarbeitung

In einem sogenannten *seriellen Rechner* existiert ein einzelner Prozessor, der die Instruktionen eines Programms einzeln nacheinander sequentiel abarbeitet. Auch wenn moderne Betriebssysteme wie Unix [UNI], Linux [LIN] oder Windows [Micb] durch *Multitasking* scheinbar eine parallele Ausführung mehrerer Programme erlauben, so wird bei einem Einprozessor-Rechner dennoch nur immer ein einziges Programm zu einem Zeitpunkt ausgeführt. Durch einen sehr schnellen Wechsel zwischen einzelnen Programmen erscheint es dem Benutzer jedoch, wie wenn seine Programme gleichzeitig ablaufen würden. Bei Rechnersystemen mit mehreren Prozessoren hingegen können die Instruktionen eines oder mehrerer Programme auf mehreren Prozessoren gleichzeitig ausgeführt werden, was in der Praxis - sofern programmtechnisch unterstützt - einen deutlichen Geschwindigkeitsgewinn verspricht.

Die zusätzlichen Prozessoren eines *Mehrprozessorsystems* können dazu verwendet werden, eine Berechnung deutlich zu beschleunigen. Im theoretischen Idealfall wird durch die Verwendung von $n$ Prozessoren eine Steigerung der Geschwindigkeit auf das $n$-fache im Vergleich zu der Geschwindigkeit auf einem Prozessor erreicht. Diese theoretisch mögliche Beschleunigung ist praktisch jedoch nicht zu erreichen, da Kommunikationsaufwand zwischen den einzelnen Prozessoren und unterschiedlichen Programmabschnitten zusammen mit inhärent seriellen Programm- und Hardware-Teilen die Beschleunigung behindert. In der Praxis können aber, trotz dieser Schwierigkeiten, immer noch Performanzsteigerungen erreicht werden, die nahe an dieses theoretische Maximum herankommen.

Um Berechnungen effizient parallelisieren zu können, ist im allgemeinen eine genaue Kenntnis des verwendeten Rechnersystems und der daraus resultierenden Anforderungen an die Programmierung erforderlich. Durch die genaue Kenntnis des Systems kann viel Rechenzeit für unnötige Kommunikation und Synchronisierung eingespart werden.

### 3.4.1 Klassifizierung Paralleler Architekturen

Um zu verstehen, wie Berechnungen parallelisiert werden können und was dabei zu beachten ist, ist es notwendig, die unterschiedlichen Architekturen paralleler Rechnersysteme zu betrachten. Aus der Architektur eines Rechnersystems folgen weitreichende Konsequenzen für seine Programmierung. Im Folgenden werden nun die wichtigsten Klassifizierungen hierzu kurz erläutert.

Ein weit verbreitetes Schema zur Klassifizierung paralleler Architekturen wurde von Flynn [Fly72] entwickelt (siehe auch Abbildung 3.10). Bei diesem Schema werden die Architekturen anhand zweier Kriterien eingeteilt:


Anhand dieser Begriffe lassen sich bestimmte Konstellationen erzeugen, die im einzelnen kurz erläutert werden.

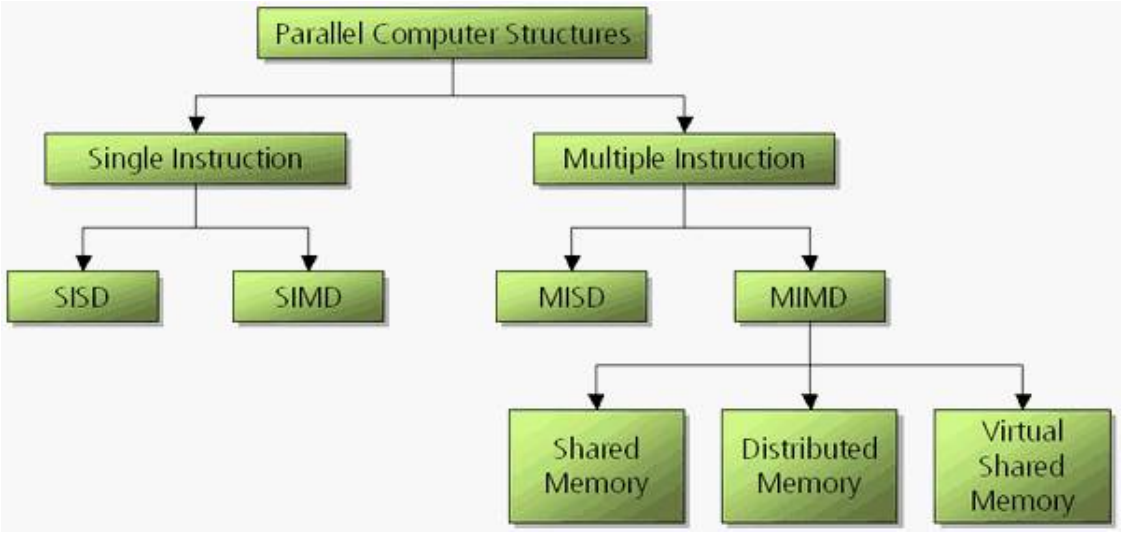



***Abbildung 3.10:*** *Klassifizierung paralleler Architekturen nach Flynn.*

**Single Instruction - Single Data (SISD)**
Hierbei wird immer eine Instruktion auf einem Datum nacheinander ausgeführt. Dies entspricht einem konventionellen seriellen Rechner.

**Single Instruction - Multiple Data (SIMD)**
Hierbei wird eine Instruktion auf mehreren Daten gleichzeitig ausgeführt. Diese Architektur wird oftmals realisiert, in dem viele einfache Prozessoren in einem großen Array zusammengeschaltet werden. Das größte Problem dabei ist, die Arbeit gleichmäßig auf alle Prozessoren zu verteilen. Ist eine gleichmäßige Auslastung der Prozessoren nicht möglich, weil zum Beispiel die Anzahl der parallel zu bearbeitenden Daten deutlich kleiner ist als die Anzahl der verfügbaren Prozessoren, dann leidet dadurch die Performanz.

Die *SIMD* Technik ist in heutige Desktop-Prozessoren übernommen worden (Intel - *MMX*, *SSE*; AMD - *3dNow*!, etc). Dabei arbeiten allerdings nicht sehr viele einfache Prozessoren dieselbe Instruktion ab, sondern es existieren eine kleine Anzahl (typischerweise 2 oder 4) parallel arbeitende Funktionseinheiten, die gleichzeitig auf unterschiedlichen Daten arbeiten. Damit wird bei geeigneten Aufgaben (z.B. 3D-Transformation) eine starke Beschleunigung gegenüber der herkömmlichen seriellen Bearbeitung erreicht, ohne die komplette Rechnerarchitektur eines Array-Rechners mit allen ihren Konsequenzen.

**Multiple Instructions - Single Data (MISD)**
Diese Architektur existiert in der Realität nicht.

**Multiple Instructions - Multiple Data (MIMD)**
Hier werden unterschiedliche Instruktionen auf verschiedenen Daten gleichzeitig ausgeführt. In diese Kategorie gehören sowohl Mehrprozessor-Systeme, als auch Cluster

von Computern, die über ein Netzwerk miteinander kommunizieren.

Diese Architektur unterscheidet man weiter nach der Art der Verbindung zwischen den Prozessoren und dem Hauptspeicher.

**Shared Memory:** Hierbei greifen alle Prozessoren auf einen gemeinsamen (globalen) Speicher zu. Der Vorteil dieser Architektur liegt darin, dass der Zugriff auf jeden Teil des Speichers von jedem Prozessor die gleiche Zeit benötigt. Außerdem ist die Programmierung relativ einfach, da keine explizite Kommunikation für den Datenaustausch zwischen den Prozessoren stattfinden muß. Nachteil dieser Architektur ist, dass sie nicht gut auf sehr viele Prozessoren skaliert. Hier ist der gemeinsame Bus zum Zugriff auf den Hauptspeicher der Flaschenhals. Dieser Flaschenhals kann bis zu einer Anzahl von etwa 32 Prozessoren noch durch Techniken, wie zum Beispiel *Crossbar Switches* [BSS00], beseitigt werden, die bei einer größeren Anzahl von Prozessoren allerdings zu aufwendig werden (siehe auch Abbildung 3.11).

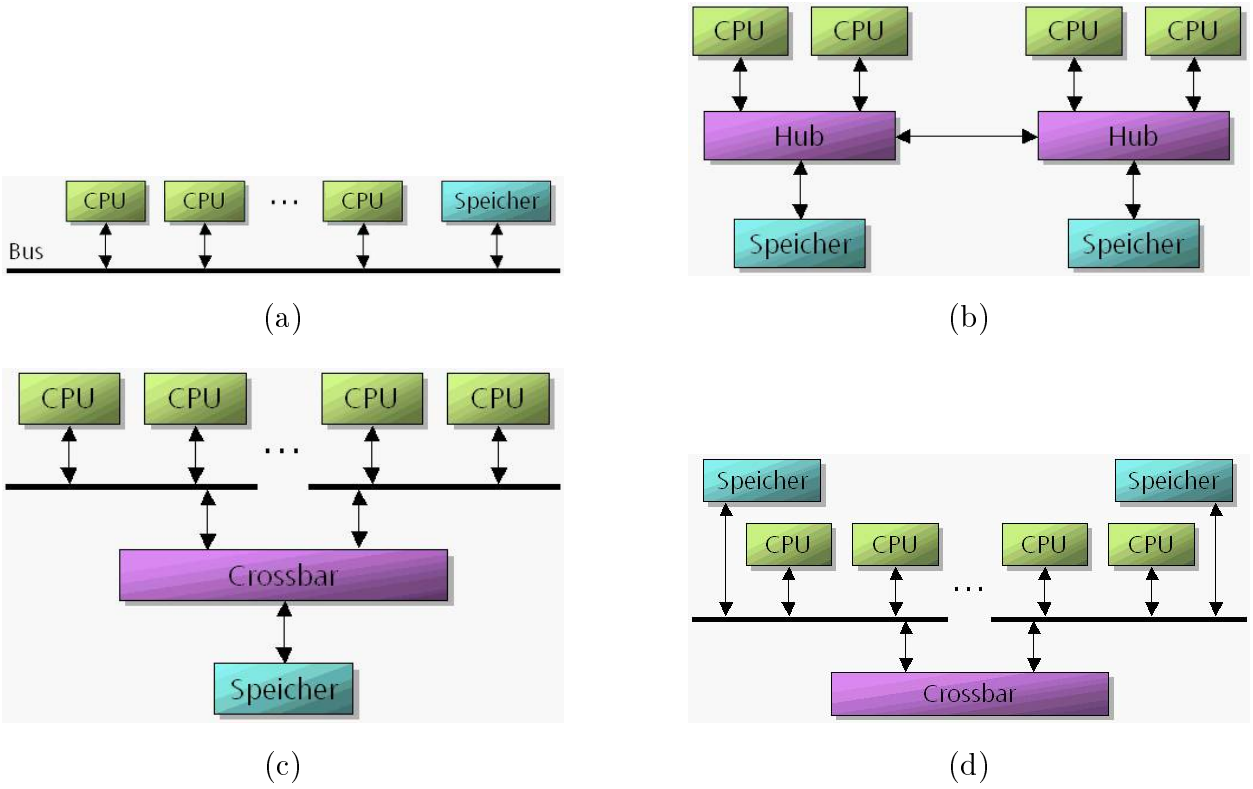



***Abbildung 3.11:*** *Verschiedene Shared Memory Realisierungen [BSS00]: (a) Standard Bus Architektur (b) SGI Origin 200 Architektur (c) HP Architektur der V-Klasse (d) Sun X500 Architektur*

**Distributed Memory:** Hier kann jeder Prozessor nur auf seinen eigenen Speicher direkt zugreifen. Um die Daten eines anderen Prozessors zu erhalten, müssen diese über explizite Kommunikation mit dem entsprechenden Prozessor angefordert werden (siehe Abbildung 3.12). Der Zugriff auf diese externen Daten dauert damit

natürlich wesentlich länger, als der Zugriff auf die lokalen Daten. Diese Architektur ist zwar aufwendiger zu programmieren, weil die Kommunikation zwischen den Prozessoren explizit erfolgen muß, dafür ist sie inhärent besser skalierbar, weil sie nicht den Flaschenhals beim Zugriff auf den gemeinsamen Speicher hat.

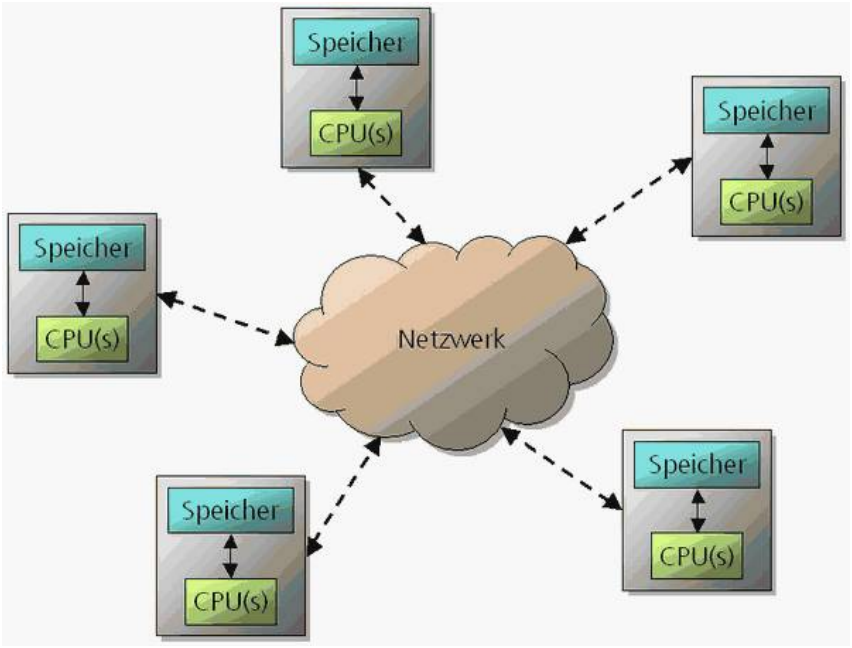



***Abbildung 3.12:*** *Distributed Memory Architektur*

**Virtual Shared Memory:** Diese Architektur ist eine Mischung aus den beiden vorangegangenen Architekturtypen. Jeder Prozessor hat einen eigenen lokalen Speicher, wie beim Distributed Memory. Zusätzlich ist aber auch der direkte Zugriff auf den Speicher eines anderen Prozessors durch die Einführung eines virtuellen globalen Adressraums möglich. Der Zugriff auf diesen externen Speicher ist aber wesentlich langsamer, als der Zugriff auf den lokalen Speicher.

## 3.4.2 Parallele Programmierungskonzepte

Im Bereich der parallelen Programmentwicklung haben sich im Laufe der Zeit zwei wesentliche Paradigmen durchgesetzt. Im Folgenden werden diese beiden Konzepte kurz vorgestellt.

### 3.4.2.1 Multithreading (Datenparallele Programmierung)

Das Hauptmerkmal beim *Multithreading* ist das Vorhandensein eines gemeinsamen Hauptspeichers, der von jedem Prozessor direkt verwendet werden kann (vgl. Kapitel 3.3.1). Die Programme sind hierbei so ausgelegt, dass unterschiedliche, voneinander trennbare Programm-Abschnitte, zeitgleich laufen können (beispielsweise die Benutzer Eingabe-Abfrage und eine

gleichzeitige Berechnung im Programm). Diese Programm-Abschnitte werden gleichzeitig gestartet und laufen in getrennten Programm-Einheiten - sogenannten *Threads*. Der globale Speicher kann hierbei sehr effizient dazu eingesetzt werden, Daten zwischen diesen unterschiedlichen Threads auszutauschen. Der große Vorteil ist, dass keine Daten zwischen verschiedenen Threads kopiert werden müssen. Es können dieselben Daten verwendet werden, solange gewisse Regeln beim Zugriff auf die Daten eingehalten werden.

Das mehrfache Lesen von Daten aus unterschiedlichen Threads ist normalerweise kein Problem. Wenn ein Thread Daten schreibt, dann muß sichergestellt sein, dass weder ein Leser gerade versucht, die Daten zu lesen, noch ein anderer Schreiber versucht, dieselben Daten zu verändern. Dieses *Leser und Schreiber Problem* [Tan92] kann beispielsweise mit der in [Tan92] vorgestellten *Semaphoren Technik* gelöst werden.

Situationen, in denen mehrere Threads auf dieselben Speicherbereiche lesend und schreibend zugreifen und deren Endergebnis davon abhängt, welcher Thread wann ausgeführt wird, werden als *Race Conditions* [Tan92] bezeichnet. Diese Race Conditions machen das Ergebnis einer Berechnung unvorhersagbar, da die Kontrolle, welcher Thread wann ausgeführt wird, nicht beim Programm selbst, sondern beim Betriebssystem liegt, und in erster Linie davon abhängt, wie viele Prozessoren zur Programm Laufzeit zu dessen Abarbeiung zur Verfügung stehen.

Das Problem bei allen Race Conditions ist der parallele Zugriff auf gemeinsame Daten von unterschiedlichen Threads. Um dieses Problem zu lösen, müssen diejenigen Stellen des Programms, die auf gemeinsame Speicherbereiche zugreifen - sogenannte *kritische Bereiche* - gegeneinander wechselseitig ausgeschlossen werden (beispielsweise ist das auch beim Zugriff auf die Festplatte der Fall). Dies kann durch den Einsatz von sogenannten *Mutex Objekten* [Tan92] realisiert werden. Ein Beispiel für den *gegenseitigen Ausschluß* (*mutual exclusion*) zweier Threads ist in Abbildung 3.13 bildlich dargestellt.

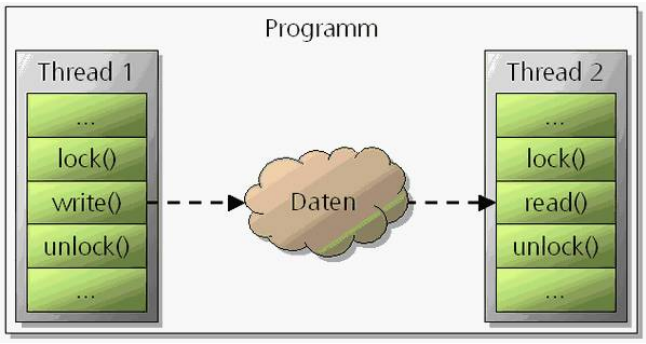



***Abbildung 3.13:*** *Mutual Exclusion: Zwei Threads, die gleichzeitig auf die selben Daten zugreifen.*

Um die Parallelität der Threads nicht unnötig einzuschränken, ist es wichtig, wirklich nur die kritischen Bereiche unter gegenseitigem Ausschluss auszuführen. In der Regel liegen nur kleine Teile der Berechnung in einem solchen Bereich. Der wesentlich größere Teil kann ohne wechselseitigen Ausschluss parallel - und damit insgesamt beschleunigt - abgearbeitet werden.

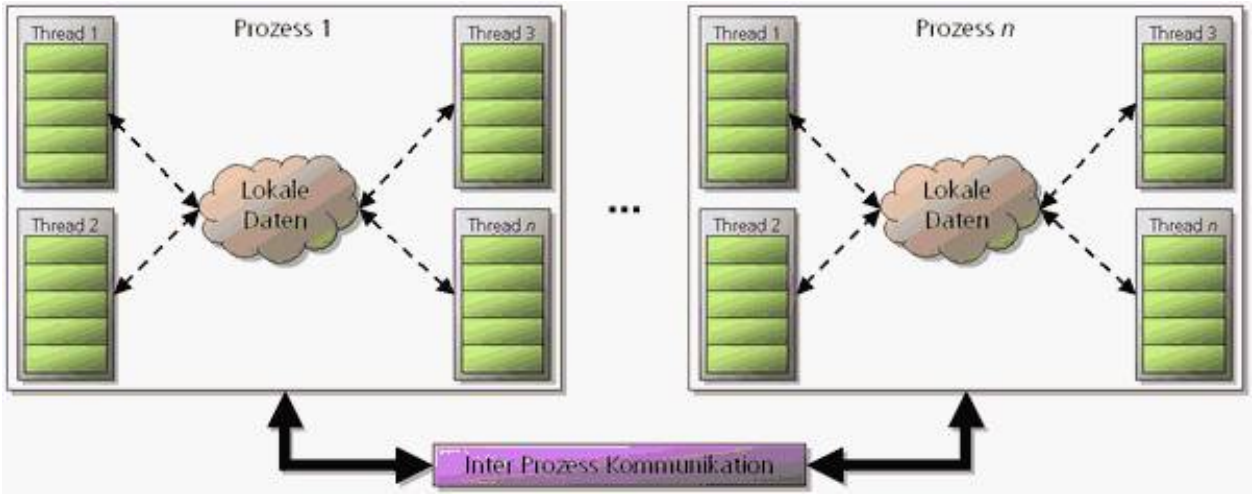



***Abbildung 3.14:*** *Inter Prozess Kommunikation*

#### 3.4.2.2 Message Passing

Zentraler Aspekt beim *Message Passing* Programmiermodell ist es, Programme bzw. *Prozesse* zur Übermittlung von Daten und zur Synchronisation untereinander Nachrichten austauschen zu lassen.

Normalerweise wird Message Passing bei Programmen eingesetzt, die nicht über einen gemeinsamen (globalen) Speicher verfügen (z.B. auch im *Cluster* Betrieb). Jeder Prozess kann dabei nur auf seinen eigenen (lokalen) Speicher zugreifen. Zugriffe auf Speicherbereiche anderer Prozesse müssen als Nachricht an den jeweiligen Prozess übermittelt werden, der dann wiederum die entsprechenden Daten als Antwort zurückschickt.

Im Gegensatz zur datenparallelen Programmierung, kann hierbei also nicht einfach von mehreren Threads auf gemeinsame Daten zugegriffen werden. Der Zugriff auf gemeinsame Daten muß vielmehr explizit formuliert und programmiert werden. Außerdem ist der Zugriff auf externe Daten durch das dazwischenliegende Kommunikations-Netzwerk wesentlich teurer als der Zugriff auf die lokalen Daten. Für eine effiziente Nutzung ist es dadurch erforderlich, dass möglichst viele der für die Berechnung benötigten Daten im lokalen Speicher vorhanden sind und nur sehr wenige Daten von externen Prozessen übermittelt werden müssen.

Diese *Inter Prozess Kommunikation* kann sowohl zwischen verschiedenen Programmen (mit getrennten Adressräumen / Speicherbereichen) als auch zwischen verschiedenen Rechnern erfolgen (Cluster) (siehe Abbildung 3.14).

Die Nachrichten, die bei dieser Inter-Prozess Kommunikation verwendet werden, werden auch als *Events* bezeichnet. Die Warteschlange, in der diese Events auflaufen, bezeichnet man als *Nachrichtenliste* oder *Event Queue* bzw. *Message Queue*.

### 3.4.3 Parallelisierungsarten

Man kann ein Programm auf verschiedene Weise parallelisieren. Im Wesentlichen lassen sich die zu unterscheidenden Parallelisierungsarten in die drei folgenden übergeordneten Kategorien einteilen ([But97, LB97, KSS96]:

**Triviale Parallelisierung:** Dies ist die einfachste Parallelisierungsart. Es wird keine der Berechnungen parallelisiert, statt dessen werden nur mehrere unabhängige Berechnungen gleichzeitig ausgeführt. Eine schematische Darstellung dieser Parallelisierungsart ist in Abbildung 3.15 zu sehen.

Diese Art der Parallelisierung hat aber auch einige gravierende Nachteile:

- Sind mehr Prozessoren verfügbar als unabhängige Berechnungen, dann können nicht alle Prozessoren genutzt werden.
- Für eine Gesamtberechnung können mehrere unabhängige Teilberechnungen erforderlich sein. Sind die Laufzeiten der Teilberechnungen sehr unterschiedlich, ist die gleichmäßige Auslastung der Prozessoren nur schwierig oder gar nicht zu erreichen.

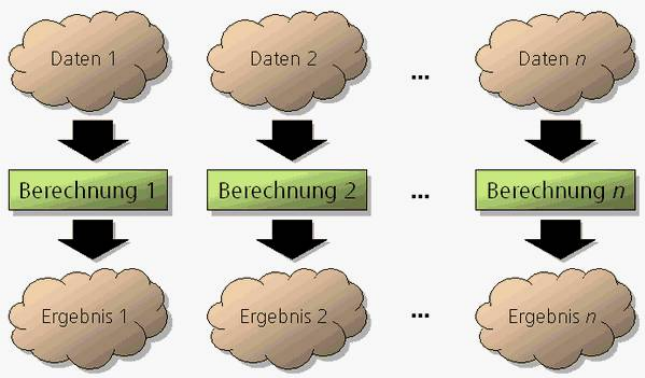



***Abbildung 3.15:*** *Triviale Parallelisierung*

**Funktionale Parallelisierung (Pipelining):** Hierbei wird eine Berechnung in Teilfunktionen zerlegt, die nacheinander ausgeführt werden. Ein Prozessor kann dann eine oder mehrere dieser Teilfunktionen übernehmen (vgl. Abbildung 3.16).

**Aufteilung der Daten:** Hier werden dieselben Berechnungen auf verschiedenen Bereichen der Daten gleichzeitig durchgeführt. Jeder Prozessor erhält einen Teil der Gesamtdaten und führt die Berechnung auf dem ihm zugewiesenen Teil aus. Die Schwierigkeit liegt hierbei in der gleichmäßigen Verteilung der Arbeit auf alle beteiligten Prozessoren (vgl. Abbildung 3.17).

Bei unbekanntem und stark schwankendem Arbeitsaufwand für Teildaten kann eine gute Auslastung erreicht werden, indem die Daten in deutlich mehr Teile zerlegt werden als Prozessoren verfügbar sind. Die Teile werden dann z.B. zufällig auf die Prozessoren verteilt, um im Mittel eine gleichmässige Verteilung der Arbeit zu erreichen.

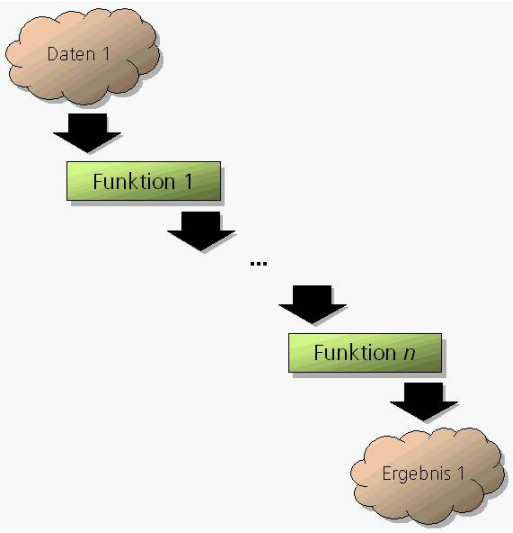



***Abbildung 3.16:*** *Funktionale Parallelisierung (Pipelining)*

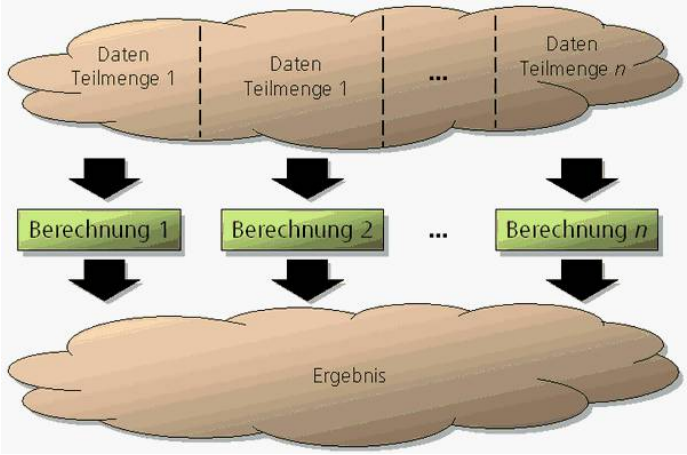



***Abbildung 3.17:*** *Aufteilung der Daten*

### 3.4.4 Performanz paralleler Programme

Zur Erfassung der Performanz paralleler Programme haben sich in Fachkreisen besonders zwei Werte als Maßstab etabliert. Hierzu wurden in [MDSH] die Begriffe *Beschleunigung* (*Speed-Up*)und die *Parallele Effizienz* definiert.

**Definition 3.4 (Beschleunigung)** *Die Beschleunigung $S$, die ein paralleles Programm durch Ausführung auf $P$ Prozessoren erreicht, ist definiert als der Quotient aus der Zeit $T(n,1)$ für die Ausführung auf einem Prozessor und der Zeit $T(n,P)$ für die Ausführung auf $P$ Prozessoren. $n$ bezeichnet hierbei die Problemgröße.*

$$S(n,P) = \frac{T(n,1)}{T(n,P)}$$

**Definition 3.5 (Parallele Effizienz)** *Die parallele Effizienz $E$ ist definiert als die Beschleunigung geteilt durch die Anzahl der Prozessoren.*

$$E(n,P) = \frac{S(n,P)}{P} = \frac{T(n,1)}{PT(n,P)}$$

Unter der Annahme, dass ein Rechner mit $P$ Prozessoren nicht mehr als $P$ mal schneller als eine Einzelprozessor Maschine rechnen kann, folgt daß

$$S(n,P) \leq P \qquad \text{und} \qquad E(n,P) \leq 1$$

Den Geschwindigkeitszuwachs, den ein paralleler Computer erreichen kann, wird durch seine inhärenten und sequentiellen Teilfunktionen begrenzt, die sich (aus technischen Gründen) nicht parallelisieren lassen.

**Satz 3.1 (Amdahls Gesetz)** *Sei $\alpha$ der Anteil der Operationen, die nicht parallelisiert werden können, wobei $0 \leq \alpha \leq 1$. Dann ist die maximale Beschleunigung eines parallelen Computers mit $P$ Prozessoren wie folgt beschränkt:*

$$S(n,P) \leq \frac{1}{\alpha + \frac{1-\alpha}{P}} \leq \frac{1}{\alpha}$$

Daraus ergibt sich beispielsweise, dass die bei einem Algorithmus, der 10% sequentiellen Code enthält, die maximale theoretische Beschleunigung durch 10 nach oben hin beschränkt ist, unabhängig von $P$, der Anzahl der verwendeten Prozessoren.

Amdahls Gesetz ist allerdings nur von Bedeutung, wenn die Problemgröße fix ist bei steigender Anzahl Prozessoren. Das ist jedoch in der Praxis selten der Fall, denn die Problemgröße tendiert dazu, mit der Anzahl der Prozessoren zu steigen, da große paralle Systeme eher dazu herangezogen werden, größere Probleme zu lösen als kleine System.

Für viele Berechnungsprobleme geht der sequentielle Anteil $\alpha$ allerdings gegen Null, wenn die Problemgröße zunimmt. Für sehr große Problemstellungen mit kleinem $\alpha$ und vielen Prozessoren spielt deswegen Amdahls Gesetz kaum noch eine Rolle und man kann Effizienzen nahe 100% erreichen.

#### 3.4.4.1 Beschränkungen der Beschleunigung

Neben dem Gesetz von Amdahl existieren noch weitere Parameter, die die Beschleunigung eines Algorithmus durch Parallelisierung begrenzen können. Diese sind beispielsweise:

- Ein optimaler sequentieller Algorithmus kann eventuell nur sehr schwer oder gar nicht parallelisiert werden.

- Um einen sequentiellen Algorithmus zu parallelisieren, ist es in der Regel notwendig, zusätzliche Operationen für das Setup der einzelnen Prozesse (z.B. Indizes etc.) und das Zusammenführen der Ergebnisse zu ergänzen. Damit steigt der Aufwand gegenüber der rein sequentiellen Variante.
- Kommunikation zwischen den parallel arbeitenden Prozessoren (zur Synchronisation oder zum Datenaustausch) erhöht den Berechnungsaufwand. Je mehr parallele Prozessoren vorhanden sind, desto mehr Kommunikation ist nötig (*Overhead*).
- Um bei der Parallelisierung eines sequentiellen Algorithmus Kommunikations-Overhead einzusparen, kann es nötig sein, Teile der Berechnung mehrfach auf unterschiedlichen Prozessoren auszuführen. Damit steigt die Gesamtzahl der für die Ausführung notwendigen Rechenoperationen.

## 3.5 Multithread Rendering

Aktuelle 3D-Graphikprozessoren werden als sogenannte *State-Machines* bezeichnet. Dieser Begriff soll die Kernfunktionalität unterstreichen, denn die von diesen Graphikprozessoren erzeugte Ausgabe hängt sowohl von ihren Eingabedaten, als auch von ihrem internen Zustand ab. Um nun aus mehreren parallelen Programmen oder Threads eine sinnvolle Ausgabe erzeugen zu können, muß für jedes Programm bzw. jeden Thread eine eigene Kopie dieses Zustandes, ein sogenannter *Graphischer-Kontext*, mitgeführt werden. Das Speichern und Wiederherstellen dieser Kontexte, zum Beispiel bei einem Prozesswechsel, wird in der Regel vom Betriebssystem übernommen. Diese Kontextwechsel müssen schnell erfolgen, damit die Ausführung der Prozesse / Threads möglichst wenig behindert wird. In der Realität zeigt sich aber, dass betriebssystembasierte Wechsel des Graphik-Kontextes im Bezug auf die Prozessorzeit recht teuer sind [BHH00] und daher vermieden werden sollten.

### 3.5.1 Graphik APIs

*Grafic Advanced Programming Interfaces* (Graphik APIs) sind Programmiersprachen-Erweiterungen, die mit speziellen Befehlssätzen den Umgang mit dem Graphikprozessor erleichtern. *OpenGL* [SGIc] und *Direct3D* [Mica] sind sogenannte *immediate mode APIs*. Sie sind jedoch nicht für die Benutzung aus mehreren Threads heraus gleichzeitig ausgelegt. Bei Verwendung dieser beiden Graphik APIs werden die (Graphik) Befehle eines (seriellen) Programms genau in der Reihenfolge, in der sie im Programm enthalten sind, an den Graphikprozessor geschickt und dort in eben dieser Reihenfolge ausgeführt. Gerne verwendet man für diese Art APIs auch den Begriff *State Machine*, da der grafische Kontext immer von einem bestimmten Zustand zum nächsten geschaltet wird.

Die unmittelbare Ausführung des jeweiligen Aufrufs auf dem Graphikprozessor ist dabei für die korrekte Funktionsweise nicht wichtig. Zur Performanzsteigerung werden die Aufrufe oftmals gepuffert und zu einem späteren Zeitpunkt gesammelt übermittelt [SA94, ISH98]. Für viele Algorithmen ist allerdings die Reihenfolge der Aufrufe wesentlich. Es muß daher

sichergestellt sein, dass die Reihenfolge, in der die Operationen auf dem Graphikprozessor ausgeführt werden dieselbe ist, wie die Reihenfolge der Aufrufe in einem Programm.

Die Funktionen eines immediate mode APIs lassen sich in drei unterschiedliche Kategorien aufteilen [BHH00]:

1. Aufrufe, die den Zustand verändern (z.B. *glBegin()*, glEnd())
2. Aufrufe, die Daten übermitteln (z.B. *glVertex3f(...)*)
3. *Sonstige Aufrufe, die weder den Zustand verändern noch Daten übermitteln (z.B.* SwapBuffers, *glClear())*

#### 3.5.1.1 Graphik Pipeline

Alle durch Graphikkommandos übermittelte Informationen durchlaufen eine Reihe von Bearbeitungstufen, die sogenannte „Graphik-Pipeline“[ogl], bevor ein entsprechender Effekt sichtbar wird. Das System puffert dabei die Aufrufe, denn einige Befehle benötigen eine gewisse Menge von Informationen bis die Aktion grafisch umgesetzt werden kann. In Abbildung 3.18 werden die einzelnen Pipeline-Stufen dargestellt, wobei die wichtigsten Aktionen jeder Stufe oberhalb der Boxen stehen. Hier wird die Funktionsweise der Graphik-Pipeline am Beispiel von OpenGL kurz vorgestellt.

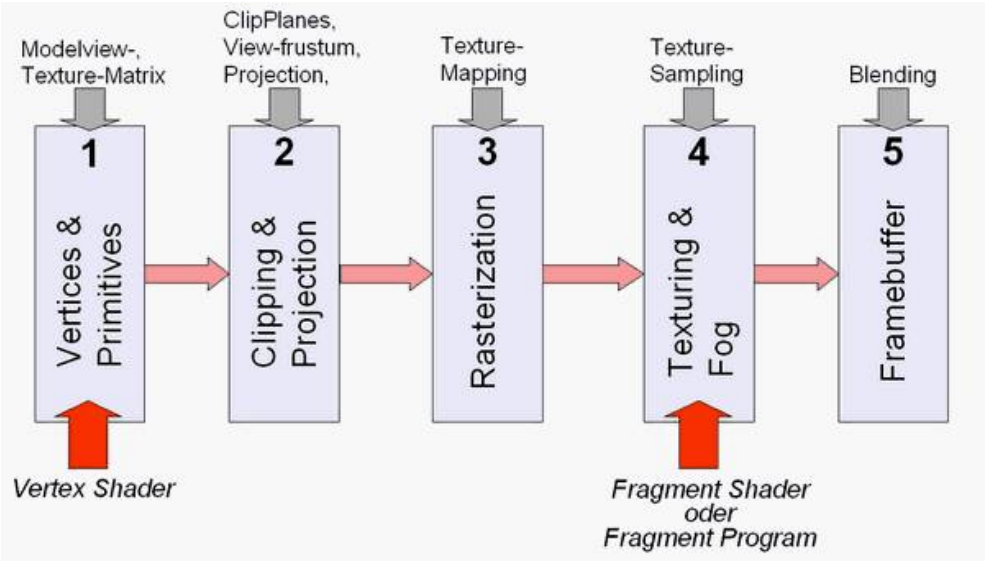



***Abbildung 3.18:*** *Schema der OpenGL Graphik-Pipeline. Die beiden Einstiegspunkte für Shaderprogramme sind durch rote Pfeile markiert. Auf den Begriff der „Shader“ wird in Kapitel 3.6.6 genauer eingegangen.*

**Vertices & Primitives**

Zwischen dem `glBegin()` / `glEnd()` Befehlspaar können Informationen für OpenGL Primitive übermittelt werden. Jede Information ist dabei an den aktuellen Knoten des

Objekts gebunden. Es handelt sich hierbei um Koordinaten (`glVertex*()`), Farben (`glColor*()`), Normalen (`glNormal*()`) und Texturkoordinaten (`glTexCoord*()`). Weiterhin verarbeitet diese Stufe per-Vertex Beleuchtung und Materialdefinitionen (`glMaterial*()`). Einige Daten dieser Stufe werden vor der Ausgabe standardmässig durch entsprechende Matrizen transformiert (*Modelview-, Texture- und Color-Matrix*) [WJS99].

**Clipping & Projection**

Ist nun eine vollständige Geometrie bekannt, werden in dieser Stufe unsichtbare Bereiche detektiert. Hierzu werden die Objekte zunächst durch frei positionierbare Schnittebenen, sog. „Clipping-Planes“, zugeschnitten. Anschliessend wird die Projektionsmatrix angewendet, welche die für perspektivische oder orthographische Projektion benötigen Transformationen durchführt. Die Positionierung der Kamera und die Einstellungen ihrer Eigenschaften definieren den sichtbaren Quader der Szene (*View-Frustum*). Alle ausserhalb liegenden Koordinaten werden deshalb durch das *View-Frustum-Clipping* erkannt und entsprechend korrigiert. Danach werden die modifizierten Koordinaten auf den Abbildungsbereich (*Viewport*) projiziert.

**Rasterization**

An dieser Stelle wird die per-Vertex Basis verlassen, um in die per-Fragment Basis überzugehen. Alle nun existiernden Primitive werden als eine Folge von Fragmenten dargestellt. Jedes Fragment verfügt über individuelle Informationen wie z.B. Farbe, Texturkoordinaten und Tiefenwerte. Die Zuweisung von Texturkoordinaten an jedes Pixel bezeichnet man als *Texture-Mapping*. Desweiteren werden in dieser Pipeline-Stufe Primitive einbezogen, die nicht durch geometrische Operationen verändert werden sollen. So z.B. Pixel-Rectangles oder Bitmaps.

**Texturing & Fog**

Das Abtasten, d.h. Sampling der Texturbitmaps geschieht durch verwenden der vorher berechneten Texturkoordinaten. Bestimmte Einstellungen für das Sampling der Textur (z.B. Interpolation oder *Padding*) werden hierbei benötigt. Padding bezeichnet das Auslesen von Koordinaten, welche über die Grösse des Speicherbereichs einer Textur hinausgehen. Hierfür kann z.B. eine Wiederholung (repeat) oder ein Limitieren der Koordinaten (*Clamping*) verwendet werden. Danach werden Einstellungen für eine Nebeldarstellung (Fog) umgesetzt.

**Framebuffer**

Der Framebuffer ist durch Bitplanes der Grösse des zugewiesenen Renderingbereichs organisiert, und liefert zu jedem Zeitpunkt die aktuelle Ausgabe des Graphikprozessors. Jede Bitplane enthält jeweils ein Bit Information für ein Pixel des Ausgabebereichs. Die Anzahl dieser Bitplanes ist je nach Aktivierung von gewünschten Fähigkeiten unterschiedlich. Durch die OpenGL Architektur werden unterschiedliche Buffer unterstützt, die verschienede Anzahlen von Bitplanes benötigen. Hierzu gehören die Gruppierungen des *Color-*, *Stencil-*, *Depth-* und des *Accumulation-Buffer*. Je nach Konfiguration werden hier auch per-Fragment Operationen ausgeführt, welche auf unterschiedliche Einträge der Buffer zurückgreifen.

Die Daten im Colorbuffer sind die Farbwerte der Farbkanäle Rot, Grün, Blau, sowie der Alphawert. Die Operationen für das Blending und den *Alpha-Test* arbeiten auf diesem Buffer. Blending ermöglicht das Mischen von Farben durch Anwendung unterschiedlicher Gewichte (Alphawert) und Verfahren, der sogenannten *Blending Function* Mithilfe des Alpha-Tests können Fragmente durch den Vergleich des Alphawertes mit einem festen Referenzwert vom Rendering ausgeschlossen werden.

Im Stencilbuffer können Bits für boolsche Vergleichsoperationen gespeichert werden, um gezielt einzelne Pixel darzustellen oder deren Darstellung zu verhindern, wodurch pixelgenaue Schablonen realisiert werden können (*Stencil-Test*).

Der Depthbuffer, oder auch *Z-Buffer*[EaSK96], ermöglicht einfache und schnelle Verdeckungsrechnung durch Verwenden des *Depth-Tests*. Je nach Wert, welcher sich bereits an einer Stelle im Depthbuffer befindet, kann durch einen Vergleich sehr einfach erkannt werden, ob sich das zu zeichnende Pixel vor oder hinter einem bereits existierenden Pixel befindet.

Mithilfe des Accumlationbuffers lassen sich mathematische Regeln für das Mischen von Farbwerten realisieren. Hierbei wird allerdings im Gegensatz zum Blending der vollständige Framebuffer beeinflusst.

Weiterhin ermöglicht diese Pipeline-Stufe das Rendering mit *Double-Buffer Technik* [EaSK96] oder weiteren Zusatz-Buffern.

#### 3.5.1.2 Szenegraph APIs

Im Gegensatz zu den oben vorgestellten immediate mode APIs werden Szenen-Graph APIs (z.B. Open Inventor [SGIb], Coin [COI] oder OpenSG [OPEc]) als *retained mode* bezeichnet. Im Unterschied zu den immediate mode APIs werden bei Szene-Graphen die Kommandos nicht direkt an den Graphikprozessor geschickt. Es wird vielmehr eine Beschreibung der Szene (in der Regel ein azyklischer, gerichteter Baum) erstellt, die dann separat traversiert und gerendert wird.

Auf diese Weise führt ein Szene-Graph eine Abstraktionsebene zwischen Applikation und Graphikprozessor ein. Dadurch muß sich die Applikation nicht mehr selbst darum „kümmern“, wie sie die darzustellenden Graphikobjekte am optimalsten an den Graphikprozessor schickt. Diese Arbeit übernimmt der Szene-Graph. Die Programmierung wird hierbei wesentlich erleichtert, da ein direktes Arbeiten auf der State Machine inklusive betreffenden Zustandsumschaltungen innerhalb eines Programmens vermieden werden.

Szene-Graph APIs sind in der Regel objekt-orientiert strukturiert. Jeder *Knoten* des Graphen ist dabei von einer Basisklasse für Knoten abgeleitet. Ein beispielhafter Szene-Graph ist in Abbildung 3.19 dargestellt. Zur Klassifizierung unterschiedlicher Aufgaben während des Rendering Prozesses, existieren verschiedene, von der Basisklasse abgeleitete Klassen. Hier werden kurz ein paar typische Vertreter vorgestellt:

**Geometrieknoten**
: beinhalten zur Darstellung notwendige Geometriedaten

**Materialknoten**
beschreiben die Materialeigenschaften darzustellender Geometrie

**Transformationsknoten**
geben Transformationen für angehängte Knoten an

**Gruppenknoten**
bieten die Möglichkeit, mehrere Knoten an einen Knoten des Graphen anzuhängen. Oftmals existieren weitere Varianten von Gruppenknoten, unter anderem:

- *Transformations-Gruppen*, die Transformationen auf alle angehängten Knoten anwenden
- *Switches*, die nur einen der angehängten Knoten nach gewissen Parametern auswählen

**Lichtknoten**
zur Beschreibung von Lichtquellen

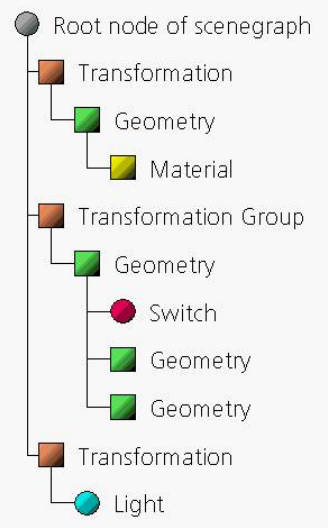



***Abbildung 3.19:*** *Szene-Graph Beispiel*

#### 3.5.1.3 Parallele Nutzung

Um nun die Probleme zu lösen, die dabei entstehen, wenn man aus mehreren Threads gleichzeitig rendern möchte, gibt es mehrere Ansätze, die teils auf immediate Mode APIs und teils auf retained Mode APIs zurückgreifen. Allen Ansätzen ist dabei gemeinsam, dass Daten von unterschiedlichen Threads nicht direkt an den Graphikprozessor übermittelt werden, sondern statt dessen eine Entkoppelung (Zwischenpufferung) der Daten implementiert wird.

#### 3.5.1.4 Pufferung von API Aufrufen

Eine relativ einfache Methode, mehreren Threads paralleles Rendern zu ermöglichen, ist, die Aufrufe des immediate mode Graphik-APIs nicht direkt auszuführen, sondern (transparent) zwischenzuspeichern. Jeder Thread hat dabei einen eigenen Kontextzustand, auf dem er unabhängig von allen anderen Threads arbeitet. Hierbei werden Rendering Primitive (Vektoren, Normalen, etc.) jedes Threads in einer Liste gespeichert, um sie später zusammen mit dem entsprechenden Zustand an den Graphikprozessor übermitteln zu können. Die Programmierung jedes einzelnen Threads erfolgt dann genauso, als ob er der einzige wäre, der auf den Graphikprozessor zugreift. Hierdurch wird zusätzlicher Programmieraufwand für den Thread vermieden.

Um die Rendering Primitive an den Graphikprozessor zu übermitteln, wird ein zusätzlicher Thread etabliert, der nacheinander die Puffer der einzelnen Render Threads ausliest und in Verbindung mit dem entsprechenden Graphik-Kontext in einer definierten Reihenfolge an den Graphikprozessor schickt. Somit wird der Graphikprozessor letztendlich doch immer nur von einem Thread direkt verwendet und dadurch gibt es keine Probleme bei der Ausführung (vergleiche Abbildung 3.20).

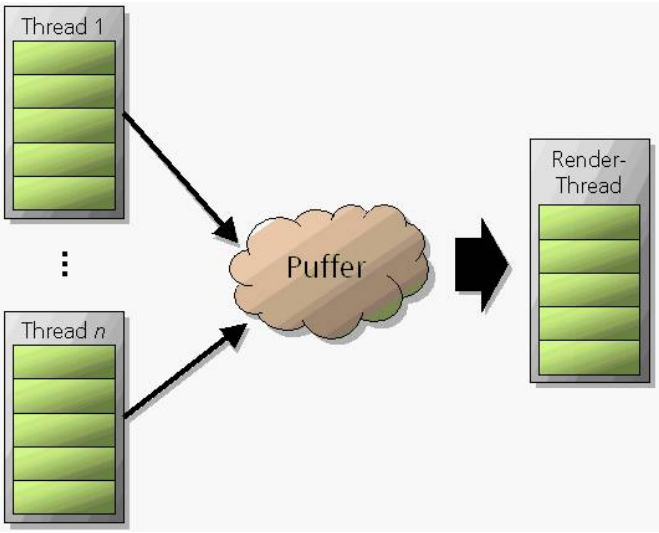



***Abbildung 3.20:*** *Pufferung von API Aufrufen und Ausgabe über einen dedizierten Render Thread.*

Der entscheidende Nachteil dieser Lösung ist es, dass es unmöglich ist, Ordnungen von Objekten über mehrere Threads hin zu erreichen, denn dafür fehlen dem Master-Rendering-Thread die notwendigen Informationen, da er nur auf Kontext- und API Befehlsebene arbeitet. Transparente Objekte aus mehreren Threads kombiniert korrekt darzustellen wird damit verhindert.

#### 3.5.1.5 Pufferung mit Synchronisationsmechanismen

In [ISH98, EIH00] wird ein paralleles Graphik Interface beschrieben, dass es zusätzlich zur Fähigkeit, Graphikaufrufe von mehreren Threads zu verarbeiten, ermöglicht, bestimmte Ord-

nungen zwischen den Operationen unterschiedlicher Threads zu definieren.

Bei den Erweiterungen des Interfaces handelt es sich um Konstrukte, die Datenströme unterschiedlicher Threads miteinander synchronisieren (z.B. *Semaphoren*). Hierbei werden nicht die Threads synchronisiert, die die Daten erzeugen, sondern die Reihenfolge der Operationen, die an den Graphikprozessor geschickt werden. Damit wird es zum Beispiel auch möglich, transparente Objekte aus unterschiedlichen Threads korrekt darzustellen.

Der entscheidende Nachteil dieses Ansatzes ist seine komplizierte Programmierung, da sich der Entwickler aktiv um die Synchronisation der einzelnen Befehle kümmern muß.

#### 3.5.1.6 Pufferung per Szenegraph

Die eleganteste Möglichkeit, paralleles Rendering aus mehreren Threads zu ermöglichen, ist die Verwendung eines *threadsafen Szene-Graph APIs*(siehe Abbildung 3.21). Ein solcher Szene-Graph ist bereits für eine Verwendung aus mehreren Threads heraus vorgesehen. Hierbei werden vom Szene-Graph API Mechanismen bereitgestellt, die sicherstellen, dass bei der Änderung des Graphen aus mehreren Threads die Daten konsistent und effizient in den Graph eingefügt werden. Es ist möglich, einzelne Knoten des Graphen zu sperren und zu bearbeiten, so daß andere Threads im Rest des Szenegraphen frei agieren können. Ein vielversprechender Ansatz in dieser Richtung ist bisher OpenSG [OPEc].

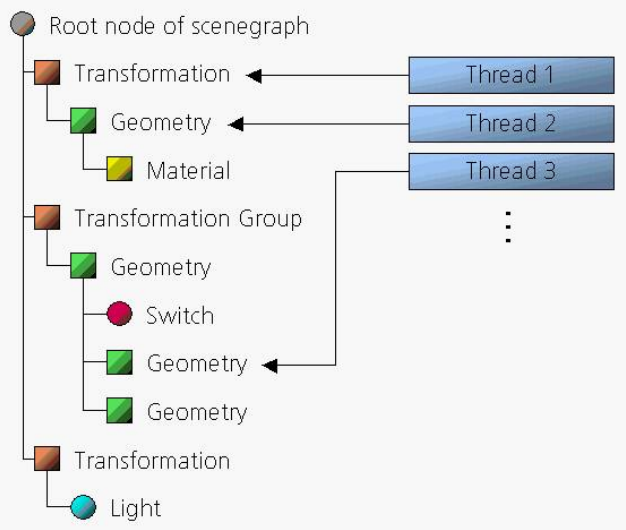



***Abbildung 3.21:*** *Pufferung von API Aufrufen und Ausgabe mit Hilfe eines Szenegraphen.*

Auch ein Szene-Graph API, das nicht für eine Benutzung mit mehreren Threads vorgesehen ist, kann durch entsprechende Synchronisationsmaßnahmen abgesichert werden. Diese Absicherung erfordert allerdings entsprechenden Programmieraufwand und die fehlende feine Kontrolle der Synchronisationsvorgänge läßt eine schlechtere Performanz als die eines für mehrere Threads ausgelegten Szenegraphen erwarten. Eine einfache Variante hierzu wäre es beispielsweise, bei jeder noch so kleinen Graphen-Änderung durch einen Thread, jedesmal

den gesamten Graphen zu sperren, die Änderung durchzuführen und den Graphen anschließend wieder freizugeben. Bei sehr vielen gleichzeitigen Änderungen am Graphen leidet die Performanz entsprechend.

## 3.6 Spezielle Grundlagen

In diesem Abschnitt werden kurz die wichtigsten technischen Grundlagen vorgestellt, die zum Verständnis dieser Arbeit nötig sind. Diese Thematiken fließen dann im Kapitel 7 in die Erläuterung der implementierten Visualisierungsmethoden mit ein (Kapitel 7.1.1 bis 7.1.5).

### 3.6.1 Phong Beleuchtungsmodell

Das Phong Beleuchtungsmodell ist ein mathematisches Modell, mit dessen Hilfe Flächen im Raum unter einer gegebenen Lichtquelle beleuchtet werden können. Hierbei entstehen Glanzlicher und dunklere Bereiche auf den virtuell beleuchteten Oberflächen. Voraussetzung hierfür ist allerdings die Kenntnis des Normalenvektors der Fläche. Siehe dazu auch [EaSK96] und [FvDH96].

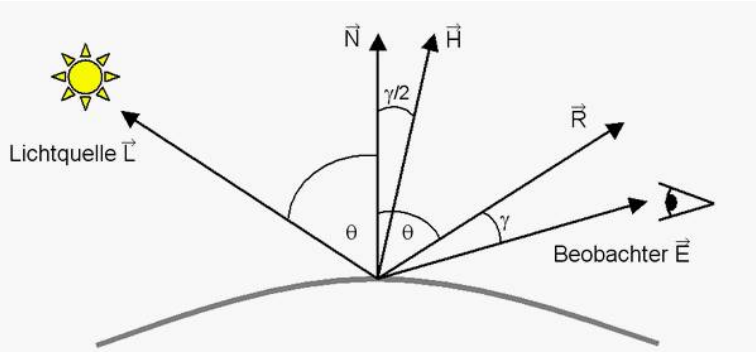



***Abbildung 3.22:*** *Geometrie des Phong Beleuchtungsmodells*

Die geometrische Situation an einer Isofläche zeigt Abbildung 3.22. Zur Berechnung der abgestrahlten Lichtintensität jedes Punktes auf der Oberfläche werden die drei empirischen Anteile aus ambienter, diffuser und spekularer Reflexion summiert.

Es gilt:

ambienter Anteil:

$$L_{ambient} = r_a \cdot L$$

ideal diffus reflektierender Anteil:

$$L_{diffuse} = \begin{cases} r_d \cdot L \cdot (\vec{N} \cdot \vec{L}), & \text{falls}(\vec{N} \cdot \vec{L}) \geq 0 \\ 0, & \text{sonst} \end{cases}$$

gerichtet diffus reflektierender Anteil:

$$L_{specular} = \begin{cases} r_s \cdot L \cdot (\vec{H} \cdot \vec{N})^m, & \text{falls}(\vec{H} \cdot \vec{N}) \geq 0 \\ 0, & \text{sonst} \end{cases}$$

mit $r_a, r_d$ und $r_s$ als ambienter, diffuser und spekularer Reflexionskoeffizient, sowie $m$ als spekularer Exponent und $L$ als Leuchtdichte des einfallenden Lichts. Dabei wird voraussgesetzt, dass alle Vektoren die Länge 1.0 haben.

Das Phong'sche Modell erhält man dann durch Summation der Lichtanteile für jede vorhandene Lichtquelle [1..n]:

$$L_{phong} = L_{ambient} + \sum_{i=1}^{n} (L_{diffuse,i} + L_{specular,i})$$

Das in diesem Konzept integrierte Phong-Beleuchtungsmodell berücksichtigt *eine* Lichtquelle mit maximaler Leuchtdichte ($L = 1.0$), die einen diffusen und spekularen Beleuchtungseffekt auf dem Objekt erzielt (im Folgenden auch als diffuse und spekulare Lichtquelle getrennt bezeichnet). Die ambienten Farbanteile $[R_a, G_a, B_a]^T$ sind dabei die vom Anwender definierten Grundfarben der Isoflächen, bzw. der klassifizierten Skalarwerte, d.h. $r_a = 1.0$. Je nach vektorieller Situation werden die Farben der diffusen Lichtquelle ($[R_d, G_d, B_d]^T$) und der spekularen Lichtquelle ($[R_s, G_s, B_s]^T$) entsprechend gewichtet und zur ambienten Grundfarbe hinzuaddiert. Bei Überlauf eines Farbkanals (z.B. $R > 1.0$), der durch die Aufsummierung entstehen kann, erhält der betroffene Kanal die maximale Intensität (*saturate*).

Die Gleichung des hier verwendeten Modells lautet:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} R_a \\ G_a \\ B_a \end{bmatrix} + r_d \cdot (\vec{N} \cdot \vec{L}) \cdot \begin{bmatrix} R_d \\ G_d \\ B_d \end{bmatrix} + r_s \cdot (\vec{H} \cdot \vec{N})^m \cdot \begin{bmatrix} R_s \\ G_s \\ B_s \end{bmatrix}. \tag{3.2}$$

Oft werden in der Literatur auch andere Bezeichnungen für die einzelnen Parameter verwendet. Hier steht z.B. $\vec{V}$ anstatt $\vec{E}$, $\alpha$ anstatt $\gamma$ oder $k$ anstatt $r$. So kann man die obige Gleichung auch in dieser abgekürtzten Form finden:

$$I = k_a + k_d(\vec{L} * \vec{N}) + k_s(\vec{V} * \vec{R})^m \tag{3.3}$$

Der optische Effekt, der durch höhere, spekulare Exponenten erzeugt wird, zeigt Abbildung 3.23.

Das Phong-Modell kann mit heutiger Shaderprogrammierung (siehe Kapitel 3.6.5) vollständig auf Hardwareebene per-pixel realisiert werden, ohne dabei vorberechnete Beleuchtungs-Texturen wie z.B. *Cube-Maps* zu verwenden. Eine Cube-Map ist ein Satz von sechs Texturen, die den Innenseiten eines Würfels entsprechen. Je nach Vektor wird die nächstgelegene Textur für das Auslesen verwendet. Steht jedoch aufgrund limitierter Hardwareressourcen keine Potenzierung als Operation im Shader zur Verfügung, können durch wiederholtes Multiplizieren Exponenten von $m = 2^i$ berechnet werden ([ati]). Dies benötigt allerdings $i$ Operationen.

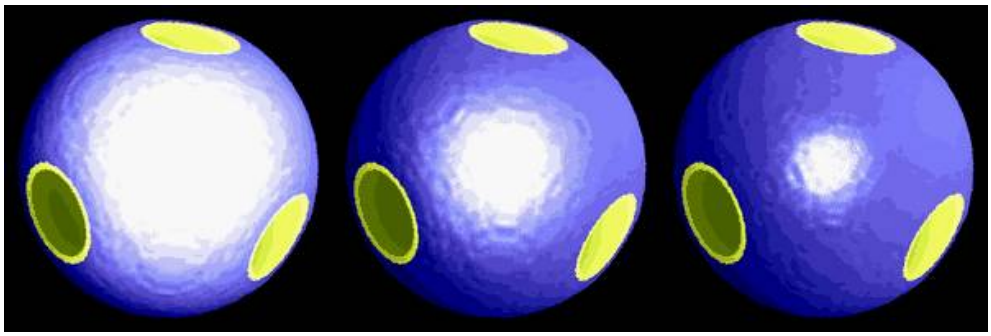


***Abbildung 3.23:*** *Spekulare Reflexion mit Exponent* $m = 4$, $m = 16$ *und* $m = 64$

### 3.6.2 Alpha Blending

Alpha Blending (englisch blend: vermischen/vermengen) ist ein Begriff der Computergraphik, der sich auf die Darstellung der Transparenz eines Bildpunktes bezieht. Die Transparenz des Bildpunktes wird neben dem Rot-, Grün- und Blauanteil im so genannten Alpha-Kanal gespeichert (vgl. Kapitel 3.5.1.1). Je nach Position und Transparenzwert des Bildpunktes wird der Farbwert mit den davor oder dahinter liegenden Bildpunkten kombiniert. Je niedriger der Transparenzwert desto durchsichtiger erscheinen dabei die Bildpunkte. Eingesetzt werden kann dieses Verfahren aber nicht nur für Transparenzeffekte, um zum Beispiel den Eindruck von Glas oder Wasser zu erzeugen, sondern auch um den Farbausdruck oder die Kontur einer Textur zu verbessern. Abbildung 3.24 zeigt mehrere semitransparente und farblich unterschiedliche Flächen einer perspektivischen Projektion eines Würfels.

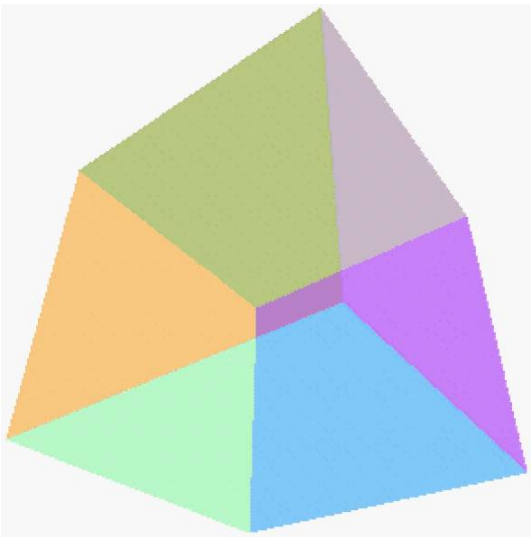


***Abbildung 3.24:*** *Alpha Blending [tec05]*

### 3.6.3 OpenGL Extensions

Da OpenGL als OpenSource Software von Silicon Graphics, Inc. [sgia] ins Leben gerufen wurde, kann jeder Entwickler die Fähigkeiten dieser Bibliothek erweitern. Neue Funktionalitäten, welche über die der aktuellsten Version hinausgehen, werden als OpenGL Extensions bezeichnet. Diese dienen meistens dazu, neue Hardware-Technologien dem Programmierer zugänglich zu machen. Daher sind viele Extensions mit dem Kürzel des Herstellers versehen, der auch die neue Technologie entwickelt hat (z.B. `NV` für NVidia Corporation [nvi] oder `ATI` für ATI Research [ati]).

Es existieren genaue Namenskonventionen und Regeln wie Extensions entwickelt werden sollen (siehe hierzu [opea]). Solange eine Funktionalität nicht in der aktuellen OpenGL Version verfügbar ist, haben alle neuen Funktionen und Konstanten (auch *Tokens* genannt) den Namensbestandteil `EXT` für „Extension“, `ARB` für *Architecture Review Board* oder eines der Hersteller Kürzel. Dabei werden `ARB`-Extensions in der nächsten Release-Version von OpenGL in den Standard Befehlssatz mit aufgenommen.

Später wird an einigen Textstellen unter anderem auf die neuesten auf heutigen Graphikkarten des Verbrauchermarktes zur Verfügung stehenden Technologien eingegangen. Hierzu gehören auch spezielle Extensions, wie z.B. `GL_EXT_texture3D`, `GL_EXT_vertex_shader`, `GL_ATI_fragment_shader` und `GL_ARB_fragment_program` ([opea]).

### 3.6.4 Texture Mapping und Multi-Texturing

Unter dem Begriff Texturen versteht man Muster bzw. Bilder, die auf Linien oder Flächen projiziert werden. In der Computergraphik unterscheidet man zwischen 1D-, 2D- und 3D-Texturen. 2D Texturen repräsentieren ein zweidimensionales Muster bzw. Bild, das auf eine Fläche aufgebracht werden kann. Der Sinn von Texturen liegt darin, eine Oberfläche nicht komplett mit farblich unterschiedlichen Primitiven beschreiben zu müssen, sondern einfach ein Bildmuster auf diese Oberfläche zu legen. Eine Oberflächenbeschreibung mit Primitiven würde zwar ein geringfügig besseres Ergebnis liefern, allerdings würde der Rechenaufwand um ein Vielfaches ansteigen. Echtzeitdarstellungen komplexer Szenen mit mehrfarbigen Flächen wären ohne Texturen mit heutigen Technologien nicht möglich.

Der grosse Vorteil der Texturierung liegt darin, dass durch das Auslesen einer Textur die Werte von der *GPU* (*Graphical Processing Unit*) bi-linear bzw. tri-linear interpoliert werden können, und damit die CPU entlasten. Weiterhin bieten sich 3D Texturen als idealer Speicherplatz für 3D Skalarfelder zur Visualisierung an.

Texture-Mapping wird i.d.R. dazu verwendet - wie beschrieben - Flächen mit Pixel-Bildern, auch *Bitmaps* genannt, zu füllen. Diese Bitmaps können z.B. eine zweidimensionale Oberflächenstruktur darstellen, welche viel feinere Details wiedergibt, als das zugrundeliegende Polygon-Netz. Dadurch ist es möglich, sehr schnell realisitisch aussehende Szenen zu rendern.

Die OpenGL Version 1.1 unterstützt lediglich 2D Texturen. Durch die Integration der Extensions ist es seit OpenGL Version 1.2 möglich, 3D Texturen zu definieren. Werden mehrere Textur-Einheiten (*Texture Units*) bereitgestellt, wird mit `glActiveTextureARB`

`(GL_TEXTURE_x_ARB)` die aktuell Gültige hiervon ausgewählt. Solange kein erneuter Wechsel der aktiven Textur stattfindet, beziehen sich alle Definitionen und Parametereinstellungen nur auf diese Textur-Einheit. Desweiteren können auch mehrere Texturen gleichzeitig verwendet werden (Multi-Texturing). Die entsprechenden Extensions liefern neue Prototypen und Tokens, mit welchen Multi-Textures und 3D Texturen definiert und bezeichnet werden können. Zur internen Verwaltung kann für jede Textur ein Name generiert werden, mit dem die Textur im weiteren Programmablauf angesprochen wird (`glGenTextures(...)` und `glBindTexture(...)`).

Sollen nun Objekte texturiert dargestellt werden, muss OpenGL angewiesen werden, in der Pipeline Stufe 4 (Texturing & Fog, siehe Kapitel 3.5.1.1) Texturkoordinaten zu generieren (rasterisieren) bzw. in Pipeline Stufe 5 (Framebuffer) auf den Texturspeicher zuzugreifen. Hierdurch hat jedes Pixel eine entsprechende Position im Textur-Bitmap. Die hierzu notwendigen Textur-Koordinaten werden der OpenGL-Pipeline auf per-Vertex-Basis durch `glTexCoord*()` bekannt gegeben. Das Koordinatensystem einer Textur ist für jede Achse auf [0.0,...,1.0] normiert, unabhängig der zugrundeliegenden Texturpixel-Auflösung.

Die Abmessungen in Pixel für eine Textur sind typischerweise auf Potenzen der Zahl 2 (d.h. 1, 2, 4, 8, 16, ...) beschränkt. Zusätzlich gibt es für die Texturen eine maximal mögliche Auflösung (z.B. 2048x2048 Pixel). Diese Maximal-Werte können je nach verwendeter Hardware und Dimension (1D, 2D, 3D) variieren. Falls die Graphikhardware die Fähigkeit der (transparenten) Textur-Kompression unterstützt, kann der tatsächlich benötigte Speicherbedarf für die jeweilige Textur geringer ausfallen als ihre theoretische Größe in Bits / Bytes. Für weitere Details bezüglich Texturen wird an dieser Stelle auf [WJS99] verwiesen.

### 3.6.5 Bump Mapping

Bump Mapping ist eine Technik, die in der Computergraphik eingesetzt wird, um eine komplexere und realistischere Darstellung unebener Oberflächen zu erhalten, ohne dabei weitere geometrische Details hinzuzufügen. Entwickelt wurde diese Technik 1978 von Jim Blinn [Bli78]. Man unterscheidet verschiedene Arten der Implementierung von Bump Mapping, hinter allen Implementierungen steckt jedoch die gleiche Grundidee: Wird eine unebene Fläche von Lichtstrahlen getroffen, so bilden sich je nach Einfallswinkel entsprechend Schatten oder Aufhellungen auf der Oberfläche (Abbildung 3.25).

Um detaillierte geometrische Strukturen von Oberflächen darstellen zu können, wird nun nicht die Komplexität des Objektes erhöht (durch mehr Polygone), sondern es werden die „Lichteffekte“, die diese Strukturen erzeugen, simuliert. Die Struktur wird dabei in einer speziellen Textur, der so genannten „Bump Map“, gespeichert und verarbeitet. Möchte man beispielsweise eine Ziegelsteinmauer möglichst akkurat darstellen, kann man die Struktur der einzelnen Fugen in einer solchen Textur ablegen.

Beim *Emboss Bump Mapping* oder auch *Multi-Pass Alpha Blended Bump Mapping* wird die Bump Map (*Height Map*) entsprechend des Lichteinfalls verschoben und von der Original-Bump Map subtrahiert (oder mit der verschobenen und invertierten Bump Map addiert). Man erhält eine so genannte *Light Map*, die noch mit der Basistextur durch Addition und

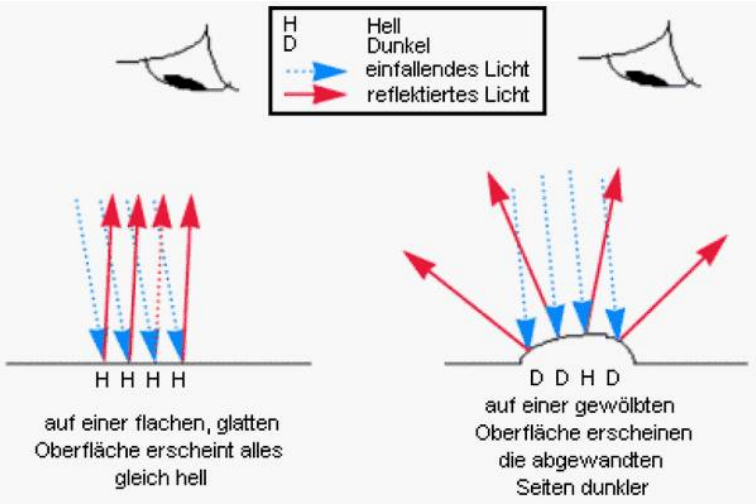



***Abbildung 3.25:*** *Lichteinfall und Schattenwurf [3dc05]*

Modulation verrechnet werden muss (Abbildung 3.26), um das gewünschte Ergebnis zu erhalten.

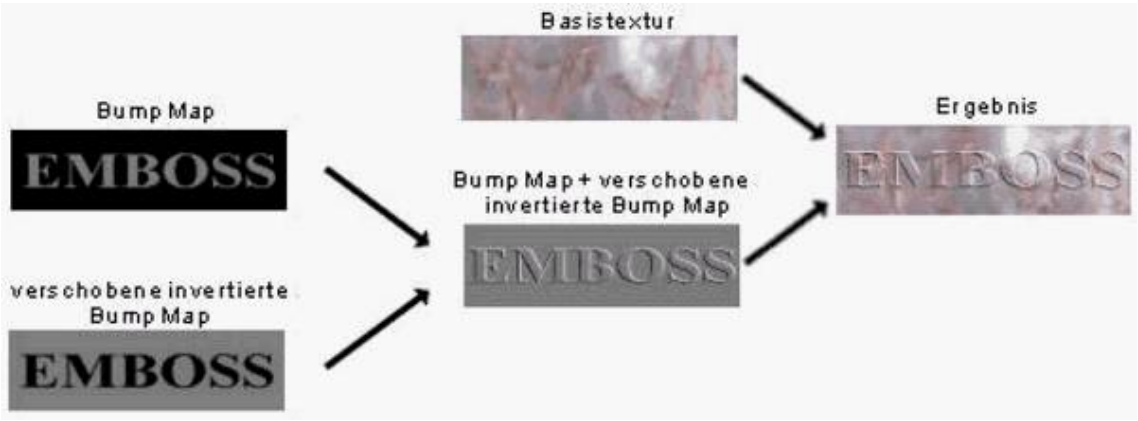



***Abbildung 3.26:*** *Emboss Bump Mapping [gam05a]*

Auch beim *Dot3 Bump Mapping* wird die Berechnung ausgehend von einer Height Map durchgeführt. Die Height Map liegt dabei im Alpha-Kanal der so genannten *Normal Map* (Abbildung 3.27) vor. In den übrigen RGB Farbkanälen der Normal Map wird eine Abweichungsnormale für den Lichteinfall an der zugehörigen Stelle gespeichert, welche sich aus den umgebenden Werten des Alpha-Kanals und dem Normalenvektor des Polygons berechnet. Zur endgültigen Darstellung wird die Abweichungsnormale mit dem einfallenden Lichtvektor verrechnet.

Neben den beiden hier vorgestellten Bump Mapping-Methoden existiert noch die Technik des *Environment Mapped Bump Mapping* [twe05], bei dem zusätzlich noch eine Umgebungstextur in die Berechnung mit einfließt.

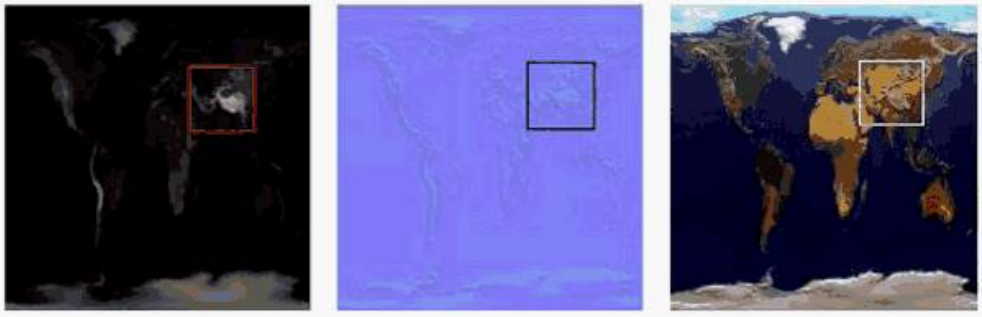


***Abbildung 3.27:*** *Height Map, Normal Map und Resultat des Dot3 Bump Mapping [gam05b]*

### 3.6.6 Shader

Die wesentlichen Einheiten beim Rendering sind, wie bereits erwähnt, die einzelnen Stufen der Graphik-Pipeline (siehe Kapitel 3.5.1.1). Jede Stufe arbeitet dabei auf einer anderen Abstraktionsebene, angefangen von Primitiven bis hin zu konkreten Pixels. Ein *Shaderprogramm* ist im Prinzip eine programmierbare Pipelinestufe, die vor deren Verwendung auf dem Graphikprozessor installiert wird (*Upload*). In Abbildung 3.18 sind die beiden verfügbaren Einstiegspunkte für Modifikationen hervorgehoben.

Der erste Einstiegspunkt wird als *Vertex Shader* oder *Vertex Processor* bezeichnet. Die Eingabe dieser Stufe sind Koordinaten von Knoten und Texturen oder verschiedenen Zustandsvariablen der OpenGL Architektur (z.B. Farben oder Lichtpositionen). Vor der Weitergabe an die nächste Stufe können mit Hilfe eines festgelegten, assemblerartigen Befehlssatzes, bzw. einer festgelegten Programmierweise alle zu diesem Zeitpunkt vorhandenen Werte modifiziert werden (*per-Vertex*). Durch den speziell von ATI [ati] vorgestellten Vertex Shader wird die standardmässige Multiplikation der Matrizen mit den Vertex-Daten außer Kraft gesetzt, was dann im Shaderprogramm berücksichtigt werden muss.

Der zweite Einstiegspunkt wird als *Fragment Shader* bezeichnet und bietet die Möglichkeit an, Fragmente kurz vor der Darstellung zu verändern (*per-Fragment*). Die zur Verfügung stehenden Eingabedaten setzen sich z.B. aus Texturkoordinaten oder Farben zusammen. Änhlich wie beim Vertex-Shader werden die standardmässigen Funktionen dieser Pipeline-Stufe durch die Aktivierung des Fragment Shaders deaktiviert, und müssen bei Bedarf durch das Shader Programm erledigt werden. Die Ausgabe ist eine neue Pixelfarbe.

Die Entwicklungen der beiden grössten Hersteller (NVidia [nvi] und ATI [ati]) hierzu, liefen eine Zeit lang parallel. NVidia Corp. arbeitet mit einer festen Anzahl von *Register Combiner*, deren Funktionsweise vom Programmierer verändert werden kann. Weiterhin ermöglichen bestimmte Befehle ein Routing von Eingabe- und Ausgabedaten, womit auch Schleifen (*Feedbacks*) realisiert werden können. In [EKE01] wurde die vorgestellte Methode zur Visualisierung von Volumendaten mit der RegisterCombiner-Technologie realisiert.

ATI Research entwickelte den Ansatz, assemblerartige Befehle zusammen mit Registern zur Verfügung zu stellen, womit kleine Programme auf per-pixel Basis ausgeführt werden können.

Diese Programmierweise setzte sich letztendlich im OpenGL Standard als `GL_ARB_vertex_program` und `GL_ARB_fragment_program` Extension durch.

### SMARTSHADER™ Vertex Shader Architektur

Abbildung 3.28 zeigt die Struktur des Vertex Shaders (`GL_EXT_vertex_shader`). Jeder Vertex kann aus bis zu 16 verschiedenen Datenanteilen bestehen (*Vertex-Streams*), wodurch jedes Datum in die implementierte Berechnung miteinbezogen werden kann. Dem Entwickler stehen dabei 12 Read-and-Write Register, sowie 96 Read-Only Register zur Verfügung. Das Shaderprogramm selbst kann aus bis zu 128 assemblerartigen Befehlen bestehen. Der Befehlssatz umfasst 24 Befehle wie z.B. arithmetische Operationen (`ADD` oder `SUB`), vektorielle Operationen (`DOT` oder `CROSS_PRODUCT`), sowie auch komplexere Operationen (`POWER` oder `RECIP_SQRT`). Hierbei wird ebenfalls zwischen den Datentypen *scalar*, *vector* und *matrix* unterschieden.

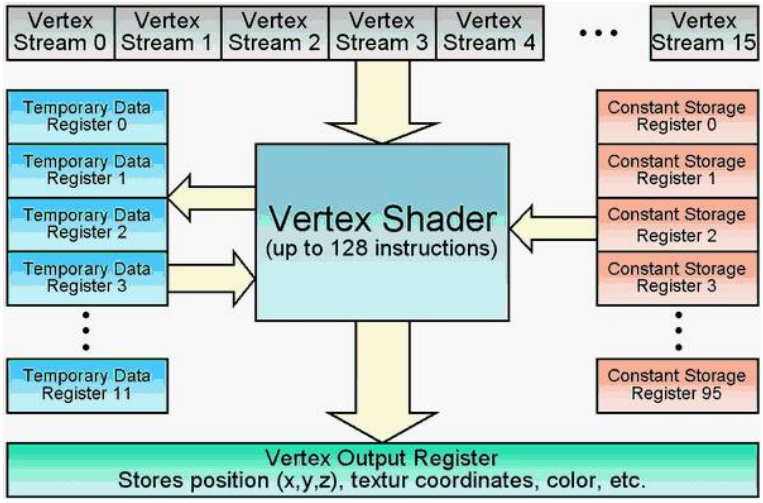



***Abbildung 3.28:*** *ATI SMARTSHADER™ Vertex Shader Architektur*

### SMARTSHADER™ Pixel Shader Architektur

Die möglichen Ein- und Ausgaben dieses Shaders (`GL_ATI_fragment_shader`) zeigt Abbildung 3.29. Es können bis zu sechs Texturen gleichzeitig gelesen, bzw. gesampled, werden. Auf dieser per-Pixel Basis wird nicht mehr zwischen Farben und Koordinaten unterschieden. Dies bedeutet, ein Vektor in einem temporären Register kann als Farbe *oder* als Texturkoordinate (*Interpolator*) verwendet werden. Dies ermöglicht die sog. *Dependent Texture Fetch* Technik, welche in dieser Arbeit einen zentralen Punkt bildet. Die Dependent Texture Fetch Technik ermöglicht es, die Texturkoordinaten eines Pixels in Abhängigkeit vorheriger Resultate von Textur-Sampling Anweisungen zu berechnen.

Die Programmstruktur unterteilt sich in zwei Pässe, vor denen jeweils texture-sampling oder register-routing durchgeführt werden kann. Der erste Pass wird als *Address-Shader* bezeichnet. Mit maximal 8 Befehlen können Texturkoordinaten vorbereitet werden, welche vor dem zweiten Pass als Interpolatoren verwendet werden. Die anschliessende Berechnung in Pass 2 muss die Ausgabe erzeugen, daher die Bezeichnung *Color-Shader*. Dieser Programmfluss ist in Abbildung 3.30 dargestellt.

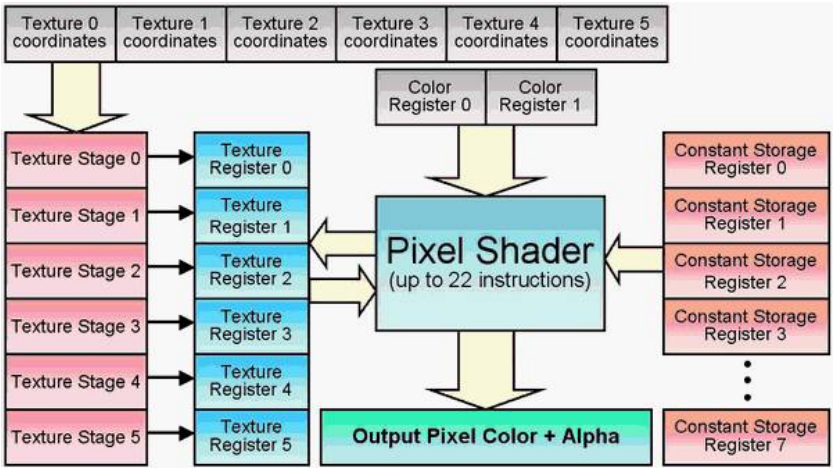



***Abbildung 3.29:*** *ATI SMARTSHADER$^{TM}$ Pixel Shader Architektur*

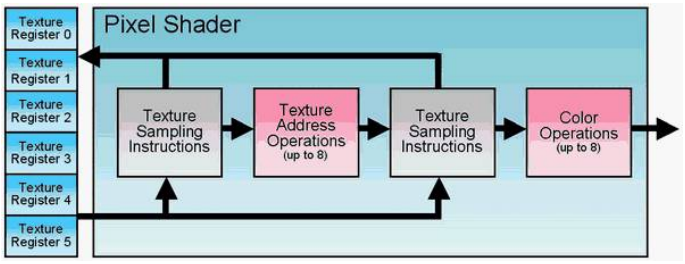



***Abbildung 3.30:*** *Programmfluss des Pixel Shaders*

### GL_ARB_fragment_program Extension

Auf Basis von `GL_ATI_fragment_shader` wurde eine einfachere und umfangreichere Extension entwickelt: die *fragment_program* Extension. Der Einsatz dieser Extension ersetzt die Funktionsweise der *fragment_shader* Extension. Durch einen Compiler bzw. Parser auf Hardwarebasis kann die GPU Programmtexte übersetzen und als Pipeline-Stufe integrieren. Der Befehlsatz umfasst 30 arithmetische Befehle und 3 Befehle für das Textur-Sampling. Ein „Pass“ (die Befehlsfolge zwischen zwei Textur-Sampling Anweisungen) wird hier als *Indirection* bezeichnet. Nun ist es wichtig zu wissen, dass die Anzahl der Indirections ausschlaggebend für die Realisierung neuer Architekturen zur Volumenvisualisierung ist. Der Umfang der Ressourcen (z.B. temporäre Variablen, maximale Anzahl von Instruktionen oder Textur-Indirektionen) ist dabei herstellerabhängig, wobei in der Spezifikation [opea] verschiedene Mindestanforderungen gestellt werden. Gerade die Resourcenmenge ist in den letzten Jahren der Hauptpunkt der Entwicklungen der Graphikkartenhersteller. Hier wurden die Karten in der vergangenen Zeit stark ausgebaut.

Als Beispiel hier einige Graphikkarten der Firma ATI [ati] und NVidia [nvi] und deren Entwicklungen im Fragment Shader (`ARB_fragment_program`) in den letzten Jahren:

### DirectX Shader 2.0/3.0 und OpenGL Shader Language

| | ATI 9x00 | ATI X8xx | ATI X1800 | Nvidia FX5xx0 | Nvidia 6800 | Nvidia 7800 |
|---|---|---|---|---|---|---|
| Max. ALU instructions | 128 | 1024 | 1024 | 1024 | 4096 | 4096 |
| Max. attribs | 10 | 10 | 10 | 16 | 16 | 16 |
| Max. env. parameters | 32 | 64 | 64 | 256 | 256 | 256 |
| Max. instructions | 160 | 1536 | 1536 | 1024 | 4096 | 4096 |
| Max. local parameters | 32 | 64 | 64 | 256 | 512 | 512 |
| Max. matrices | 32 | 32 | 32 | 8 | 8 | 8 |
| Max. matrix stack depth | 16 | 16 | 16 | 1 | 1 | 1 |
| Max. native ALU instructions | 64 | 512 | 512 | 1024 | 4096 | 4096 |
| Max. native attribs | 10 | 10 | 10 | 16 | 16 | 16 |
| Max. native instructions | 96 | 1024 | 1024 | 1024 | 4096 | 4096 |
| Max. native parameters | 32 | 64 | 64 | 1024 | 1024 | 1024 |
| Max. native temporaries | 32 | 64 | 64 | 32 | 32 | 32 |
| Max. native texture indirections | 4 | 4 | 4 | 1024 | 4096 | 4096 |
| Max. native texture instructions | 32 | 512 | 512 | 1024 | 4096 | 4096 |
| Max. parameters | 32 | 64 | 64 | 1024 | 1024 | 1024 |
| Max. temporaries | 46 | 128 | 128 | 32 | 32 | 32 |
| Max. texture coords | 8 | 8 | 8 | 8 | 8 | 8 |
| Max. texture image units | 16 | 16 | 16 | 16 | 16 | 16 |
| Max. texture indirections | 31 | 511 | 511 | 1024 | 4096 | 4096 |
| Max. texture instructions | 32 | 512 | 512 | 1024 | 4096 | 4096 |

***Tabelle 3.1:*** *Graphikkarten Fragment Programm-Kapazitäten [del05]*

Gerade im Bereich der Shaderentwicklung finden im Moment sehr große und weitreichende Entwicklungen statt. Während die bisher vorgestellten Technologien gewissermaßen den Einstieg in die programmierbare Graphikpipeline darstellen, geht die Entwicklung ständig weiter. Neben den über OpenGL zugänglichen Shader Technologien ist Microsoft mit der in Direct3D/DirectX [Mica] integrierten Shader Technologie 2.0/3.0 [mss05] auch ein wichtiger Motor, der die zugehörige Forschung vorantreibt. In aktuellen Computerspielen mit integrierter 3D Szenerie und Darstellung werden bereits viele dieser Techniken praktisch umgesetzt. Bezüglich OpenGL gibt es bereits mehrere Ansätze, eine *Shading Language*, d.h. eine standardisierte Programmiersprache für die Shader Implementierung zu etablieren. Mittlerweile ist auch hier eine Standardisierung erfolgt [ogl05]. In diesem Bereich sind in den nächsten Jahren noch deutliche Weiterentwicklungen und Forschungen zu erwarten.

## 3.7 Zusammenfassung

Zu Beginn dieses Kapitels wird zunächst der Begriff der Visualisierungsmethode definiert. Er beschreibt im Zusammenhang mit dieser Arbeit den Vorgang, der aus vorgegebenen Strömungssimulationsdaten eine interpretierbare Darstellung errechnet. Die Begriffe der technisch-wissenschaftlichen und der realitätsnahen Visualisierung werden definiert und unterschieden.

Im Anschluß daran erfolgt eine wissenschaftliche Einordnung der Thematik. Im Gebiet der parallelen Datenverarbeitung haben sich seit geraumer Zeit verschiedene Ausprägungen entwickelt und durchgesetzt. Im Bereich der Simulation von Strömungen im Rechner werden parallelisierte Verfahren schon seit längerer Zeit eingesetzt. Im Bereich der Visualisierung etablieren sich nunmehr auch parallelisierte Ansätze. Hierbei kann die Parallelisierung sowohl auf Hardware - als auch auf Softwareseite stattfinden. Mit Hilfe intelligenter Pufferungsmethoden gelingt es durch den Einsatz von sogenannten „threadsafen“ Szenegraphen, die optimierte Ausgabe der Visualisierungsdaten von den dafür notwendigen Berechnungen abzukoppeln. Letztendlich zerfällt dadurch das Problem der (parallelisierten) Visualisierung von wissenschaftlichen Simulationsdaten in 2 Bereiche, die getrennt voneinandern behandelt werden können. Auf der einen Seite steht die hochoptimierte Ausgabe der Visualisierungsdaten, die durch Filterung und Analyse der Simulationsdaten entstanden sind. Teile dieses Prozesses sind heute bereits in Hardware verfügbar (Graphik-Pipelines) und werden durch das Betriebssystem abgedeckt („transparente“ Cluster wie Mosix [MOS]) oder von Szenegraphen (im Hintergrund) erledigt. Auf der anderen Seite stehen die Vorberechnungen für die Visualisierung. Hier wird die Ergebnisdatenmasse der Simulationsprogramme analysiert, gefiltert und deren Ausgabe vorbereitet. Es werden entsprechende Geometrie-Primitive erzeugt und im Speicher abgelegt (Dreiecke, Texturen, Punkte, Farbtabellen). Auf diesem Gebiet bestehen noch Entwicklungspotentiale, die die volle Ausschöpfung der verfügbaren Möglichkeiten beispielsweise durch Parallelisierung und/oder Nutzung der neuesten Möglichkeiten aktueller Graphikhardware erlauben, ohne sich dabei auf ein konkretes Hardwaremodell oder konkrete Datenformate festlegen zu müssen. Dabei werden folgende Aussagen gemacht:

- Strömungssimulationen erzeugen verfahrensbedingt große Datenmengen. Die Simulationsanwendungen arbeiten zumeist parallelisiert und schreiben die Ergebnisdaten in spezifischen Dateiformaten heraus.

- Im Gebiet der Strömungsanalyse haben sich eine Reihe von wissenschafltichen Visualisierungsmethoden etabliert, mit denen es den Forschern heutzutage möglich ist, die für sie wichtigen Informationen aus der Ergebnisdatenmenge herauszufiltern.

- Es existieren auf dem Softwaremarkt eine Reihe von Strömungssimulations-Werkzeugen. In diese Werkzeuge sind oft kleine Visualisierungsanwendungen integriert, die jedoch in Punkto Qualität und Geschwindigkeit bei weitem nicht so leistungsfähig sind, wie es heutzutage bereits möglich ist.

- Gleichzeitig sind zwar (interaktive) Visualisierungswerkzeuge verfügbar, die eine umfangreiche Bibliothek an Visualisierungsmethoden bieten, jedoch arbeiten diese meist sequentiell oder sie sind hochspezialisiert und auf bestimmte Datenformate optimiert.

- Die wenigen verfügbaren parallel arbeitenden Programme für die Visualisierung von Strömungsdaten, arbeiten i.d.R. nach dem Raytracing Prinzip und sind aufgrund dessen (noch) nicht echtzeitfähig, d.h. interaktionsfähig. Sie sind nicht so angelegt, dass sie schnell und einfach um neue (parallelisierte) Visualisierungsverfahren erweitert werden können, die die aktuelle Darstellungsqualität und Geschwindigkeit der Graphikhardware nutzen.

- Die auf dem Markt erhältlichen Visualisierungswerkzeuge haben neben der Parallelisierung keine weiteren Konzepte für die Unterstützung großer Simulationsdatenmengen, die beispielsweise über die Größe des zur Verfügung stehenden Arbeitsspeichers hinausgehen.
- Keine der verfügbaren Softwarepakete zur Darstellung von Strömungsdaten mit Hilfe von technisch-wissenschaftlichen Visualisierungsmethoden unterstützt zusätzlich die realitätsnahe Darstellung (entsprechende Visualisierungsmethoden und Umgebungsgeometrie).
- Mittel zur Steuerung des Datenbereiches, in dem die aktivierten Visualisierungsmethoden aktiv sind, sind in den meisten erhältlichen Programmen nicht vorhanden. Die Visualisierung arbeitet statt dessen immer auf dem gesamten Datenraum.

Im Anschluß werden verschiedene Speicherdatenformate für Strömungsdaten vorgestellt. Sie unterscheiden sich bezüglich Struktur und Aufbau, z.B. anhand der verwendeten Sortierung oder Zelltypen.

Nach einem Ausflug zu den Grundlagen der Parallelen Datenverarbeitung wird insbesondere auf die Probleme des Multithread Renderings eingegangen. Hierbei wird die klassische Graphik Pipeline untersucht und deren Erweiterung durch Szenegraphen, die Ausgabekommandos puffern können, vorgestellt.

Zum Abschluss des Kapitels wird noch auf spezielle technische Grundlagen eingegangen, die zum Verständnis dieser Arbeit wichtig sind. Hierzu gehört beispielsweise die Erläuterung des Phong Beleuchtungsmodells, als auch die Vorstellung der OpenGL Extensions oder die moderne Shader Technologie aktueller Graphikkarten.

# Kapitel 4

# Problemanalyse und Anforderungen

In diesem Kapitel werden die unterschiedlichen Problemstellungen, die sich im Zusammenhang mit dem Thema dieser Arbeit ergeben, genau analysiert und aus ihrem Kontext heraus die entsprechenden Anforderungen an ein System zur massiv parallelen Visualisierung großer Datenmengen formuliert. Im ersten Unterkapitel sind hierbei die konzeptionellen Anforderungen an ein Visualisierungssystem zusammengefaßt. Daran anschließendend wird dann auf die technischen Anforderungen genauer eingegangen.

## 4.1 Konzeptionelle Anforderungen

Möchte man die Anforderungen an ein Visualisierungssystem für Strömungsdaten genauer beschreiben, so lassen sich verschiedene konzeptionelle Bedingungen finden, die man als Voraussetzung für ein solches System definieren kann. Hierbei dreht es sich um prinzipielle Vorgehensweisen und Abläufe, die Vorraussetzungen eher an die Methodik als an die technische Umsetzung richten. In diesem Unterkapitel wird auf diese Anforderungen genauer eingegangen.

### 4.1.1 Kompression

Strömungs-Simulationssoftware arbeitet heutzutage mit zeitlich und räumlich diskretisierten Datenfeldern: Im Zuge der Simulation einer vorgegebenen Situation (z.B. ein Brand in einem Tunnel), werden die genauen Umgebungsbedingungen (wie z.B. Ausgangstemperatur, Druck, Wetter und Windbedingungen, brennbare Materialien, etc.) und die in der Szene befindlichen Geometrien (Gebäude, Fahrzeuge, etc.) auf verschiedene Parameter abgebildet. Anschließend startet die Simulation ihre Berechnungen. Dieser Vorgang läuft typischerweise so ab, dass der gesamte Simulationsraum in ein Datengitter diskretisiert wird. Die Berechnung neuer Zustände wird nur für diese diskreten Datenpunkte durchgeführt. Wenn ein neuer Berechnungszustand erreicht ist, spricht man von einem neuen „Zeitschritt“. Von diesem neuen Datensatz ausgehend startet die Simulation erneut und rechnet auf dieser Grundlage

weiter zum nächsten Zeitschritt. Die Speicherung der Daten auf die Festplatte erfolgt nun typischerweise nach jedem dieser Simulations Zeitschritte oder nachdem eine bestimmte Anzahl von Schritten errechnet wurde. Dabei wird jedesmal der komplette Datenraum als Datei auf der Festplatte ablegt. Bei hinreichend feiner Diskretisierung hinsichtlich Raum und Zeit und bei entsprechender Länge des simulierten Zeitabschnittes, kommen hierbei schnell sehr große Datensätze zusammen, die weit über die Größe des Arbeitsspeichers des verwendeten Rechners hinausgehen.

In den verwendeten Werkzeugen zur Simulation von Strömungen finden primär keine speziellen Methoden zur Reduktion dieses Datenvolumens (Kompression) Anwendung. Das liegt darin begründet, dass der Fokus bei diesen Applikationen in erster Linie darauf ausgerichtet ist, die Simulation möglichst schnell zu machen. Hierbei bedeutet die reine Speicherung der Daten an sich schon einen Zeitverlust, da sie jedesmal durchgeführt werden muss, bevor die Simulation weiterrechnen kann. Wenn an dieser Stelle nun noch eine aufwendige Komprimierung der Daten stattfinden würde, dann wäre zusätzliche Speicherzeit von Nöten, die sich bei entsprechend vielen Zeitschritten schnell drastisch auf die Laufzeit der Simulation auswirken würde. Genauso hinderlich wäre es auf der anderen Seite, wenn die Simulationssoftware beim Zurücklesen der Daten diese erst zeitraubend entpacken und interpretieren müsste. Dieser Zeitfaktor ist mit einer der Hauptgründe, warum von Seiten der Simulationssoftware-Entwicklung in erster Linie wenig Aufmerksamkeit auf das Thema Datenkompression gerichtet wird. Ein radikaler Ansatz für dieses Problem beschränkt sich lediglich darauf, bei der Speicherung der Daten eine gewisse Anzahl von Zeitschritten auszulassen, d.h. sie zwar zum Weiterrechnen zu verwenden, aber nicht abzuspeichern. Es wird beispielsweise nur jeder zehnte Zeitschritt gespeichert. Die Genauigkeit zwischen den gespeicherten Schritten geht der Visualisierung im Anschluss verloren.

Möchte man nun allerdings diese Datenmengen auf dem Bildschirm darstellen, stößt man sehr schnell auf das Problem, dass man zum einen sehr viele verschiedene Zeitschritte bewältigen können muss, wenn die Simulation einen großen Zeitraum umfaßt. Zum anderen muss die Visualisierung mit sehr großen Datengittern bzw. feinen räumlichen Diskretisierungen umgehen können, denn die Visualisierungsmethoden arbeiten in der Regel so, dass sie die Daten eines vorliegenden Zeitschrittes interpretieren und daraus eine grafische Darstellung errechnen. Für die meisten Visualisierungsmethoden ist es sehr wichtig, dass sie für ihre Berechnungen Datenwerte aus dem gesamten Simulations-Datenraum abfragen können. Bei einigen Visualisierungsmethoden ist es sogar notwendig, dass sie für die dazugehörigen Berechnungen zusätzlich auch noch mehrere Zeitschritte gleichzeitig benötigen. Es würde also nicht genügen, einfach nur bestimmte Teile des untersuchten Datensatzes in den Speicher zu laden, da der gesamte Datenraum und gegebenenfalls auch noch mehrere Zeitschritte gleichzetig benötigt werden. Probleme entstehen, wenn die damit verbundene Datenmenge zu groß ist, um sie komplett in den Speicher des Rechners laden zu können, sie also zu groß ist, um auf ihr schnell und effektiv lesende Zugriffe für die Berechnungen der Visualisierung durchführen zu können. Die Anforderung, die sich aus diesem Kontext ergibt, ist die Kompression der Daten für die Visualisierung. Hier wird ein Ansatz benötigt, der auch auf Rechnern, die mit weniger Speicher bestückt sind, eine adäquate Visualisierung der Simulationsdaten ermöglicht.

### 4.1.2 Multivariate Visualisierung

Um die von der Simulation errechneten Daten zu analysieren, bedient sich die Darstellung der Visualisierungsmethoden (siehe Kapitel 3.1). Durch sie ist es möglich, bestimmte Merkmale des Datenraumes zu extrahieren und darzustellen. Möchte der Nutzer die Ergebnisse zweier Visualisierungsmethoden miteinander vergleichen, so hat er zwei Möglichkeiten. Zum einen kann er die Ergebnissbilder der beiden Darstellungen abspeichern und diese Bilder miteinander vergleichen. Eine andere Möglichkeit ist es, einfach beide Visualisierungsmethoden, gleichzeitig zu aktivieren und deren Ergebnisse in einem Bild zu vereinen. Da es bei mehreren gleichzeitig aktivierten Visualisierungsmethoden schnell zu Bereichen kommt, in denen die eine Darstellung die andere überlappt, gibt es demzufolge Teile, die verdeckt und nicht direkt einsehbar sind (Abbildung 4.1). Die gleichzeitige Darstellung mehrerer Visualisierungsmethoden im Zusammenhang mit dieser Arbeit wird als *Multivariate Visualisierung* bezeichnet.

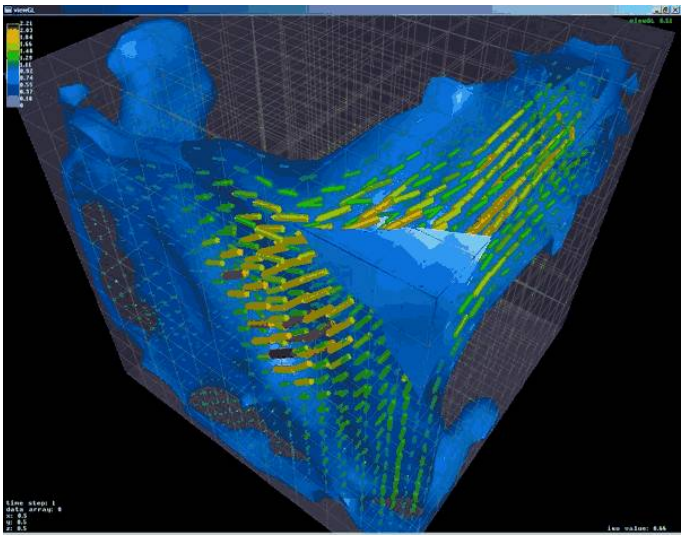


***Abbildung 4.1:*** *Gleichzeitige Darstellungen des Geschwindigkeitsfeldes (Vektorpfeile), des Geschwindigkeitsbetrages (Farbe und Größe der Vektorpfeile) und des Temperaturfeldes (Iso-Fläche) einer Strömungssimulation.*

Um dieses Problem der Überdeckung zweier Visualisierungsmethoden zu lösen, wird die Möglichkeit benötigt, die „Durchsichtigkeit“ bzw. *Transparenz* einer Visualisierungsmethode steuern zu können.

Eine dritte Möglichkeit, die Darstellung zweier Visualisierungsmethoden miteinander zu vermischen, ist es, wenn die Ergebnisse der einen Visualisierungsmethode die Darstellung einer zweiten Methode beeinflussen. Hier haben die Werte, die aus der Berechnung der ersten Methode kommen direkten Einfluß auf die Visualisierung der zweiten (Abbildung 4.1 (die Werte des Geschwindigkeitsbetrags-Skalarfeldes beeinflußt die Einfärbung und Größe der Pfeilvisualisierung des Geschwindigkeits-Vektorfeldes) und Abbildung 4.2).

Eine wichtige Anforderung, die sich nun aus diesen Zusammenhängen ergibt, ist die, dass es in einem Visualisierungssystem möglich sein sollte, mehrere Visualisierungsmethoden gleich-

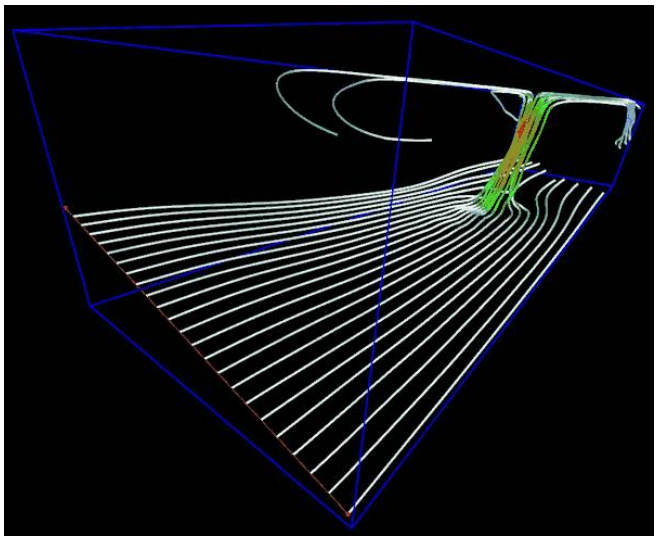


***Abbildung 4.2:*** *Anhand des Temperaturfeldes eingefärbte Strömungslinien.*

zeitig darstellen zu können (Multivariate Visualisierung). Eine weitere Bedingung, die sich in Anlehnung hieraus ergibt, ist die, dass es möglich sein muß, die Transparenz jeder Visualisierungsmethode steuern zu können. Sie muss durchscheinend darstellbar sein, um durch sie verdeckte Elemente / andere Visualisierungsmethoden sichtbar zu machen.

### 4.1.3 Realitätsnahe Darstellung der Umgebung

Heutzutage ist Software, die einem Anwender im Bereich der Strömungs-Simulation zur Verfügung steht, sehr weit entwickelt. Zusammen mit der immer weiter steigenden Leistungsfähigkeit der Hardware ist es mittlerweile leicht möglich, sehr komplexe räumliche Zusammenhänge in der Simulation weitestgehend korrekt abzubilden, ohne dass man starke Vereinfachungen machen muss. Musste beispielsweise früher ein Zimmer zu einem groben Quader für die Simulation vereinfacht werden, da die Leistungsfähigkeit der Hardware und Software für einen komplexeren Sachverhalt nicht ausgereicht hat, so ist es mittlerweile ohne weiteres möglich, das komplette Innenleben des Zimmers (Möbel, Ausbuchtungen, Luftabzüge) in der Simulation abzubilden und bei den Berechnungen mit zu berücksichten. Will man beispielsweise den Luftzug in einer U-Bahn Station simulieren, so ist man mittlerweile dazu fähig, das komplette Lüftungssystem inklusive der vollständigen Umgebungsgeometrie in der Simulation abzubilden und auf dieser Datengrundlage zu simulieren.

Möchte man nun in diesen recht komplexen Strukturen die errechneten Daten simulieren, so wird es immer wichtiger, neben den reinen Ergebnissen, die per Visualisierungsmethode dargestellt werden, auch die Umgebung selbst mit auszugeben. So hat der Nutzer die Möglichkeit, korrekt und intuitiv in der modellierten Umgebung zu navigieren und sich schnell zurecht zu finden und kann die dargestellten Daten schnell den entsprechenden Lokalitäten (beispielsweise ein Abzugsschacht in einem Lüftungssystem) zuordnen. Eine reine Präsentation der Ergebnisse der Visualisierungsmethoden ohne die Umgebungsgeometrie ist in den

meisten Fällen aufgrund der Komplexität der Szene nicht mehr ausreichend.

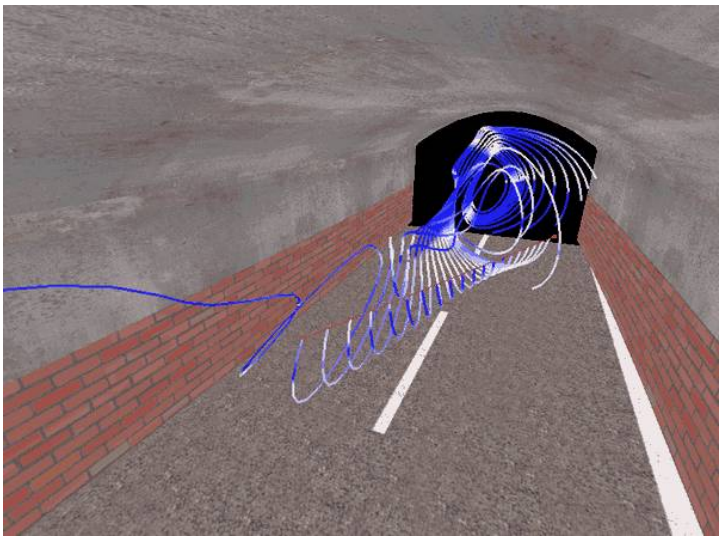


***Abbildung 4.3:*** *Realitätsnahe Darstellung der Umgebung (Tunnel) zusammen mit einer technisch-wissenschaftlichen Visualisierung (Strömungslinien).*

Neben einer rein technischen Darstellung der Umgebung, beispielsweise als einfarbige Flächen, wird es aufgrund der steigenden Komplexität der Szenen für den Wissenschaftler immer interessanter, diese Umgebung möglichst realistisch zusätzlich einzublenden. Hier spielt beispielsweise die Visualisierung der Farbe und Beschaffenheit von Oberflächen oder eine Schattierung derselben eine Rolle. Durch eine möglichst realitätsnahe Darstellung der Simulationsumgebung wird auch dem Simulations- und Visualisierungs-Laien schnell ersichtlich, in welchem Zusammenhang die visualisierten Daten stehen. Die Anforderung, die sich in diesem Kontext ergibt, läßt sich damit beschreiben, dass es möglich sein muss, neben der Darstellung der Visualisierungsmethoden an sich auch die Umgebungsgeometrie darzustellen. Zur besseren Interpretierbarkeit soll diese Umgebung realitätsnahe (d.h. mit korrekten Farben und Beleuchtung) geschehen (Abbildung 4.3). Wobei es hier trotzdem sinnvoll ist, gewisse Grenzen zu berücksichtigen. Die Ausgabe der Umgebungsgeometrie ist zwar gewünscht und hilfreich, steht jedoch nicht im Vordergrund. An dieser Stelle ist es also nicht nötig so weit zu gehen, die neuesten Rendering Techniken einzubinden, um quasi fotorealistisches Rendering zu ermöglichen - auf Kosten der Darstellungsleistung und Rechenzeit der eigentlichen Visualisierungsmethoden.

Bezieht man diese Eigenschaft der realitätsnahen Darstellung nicht nur auf die Umgebungsgeometrie, sondern auch auf die Visualisierungsmethoden selbst, so kommt man zur nächsten Anforderung (siehe nächstes Kapitel).

### 4.1.4 Technisch-wissenschaftliche und realitätsnahe Visualisierung

Wie in Kapitel 3.1 beschrieben, finden neben rein technisch-wissenschaftlichen Visualisierungsmethoden auch realitätsnahe Methoden ihre Anwendung. Interessante Darstellungen

lassen sich erzielen, wenn man in einem System beide Darstellungsarten kombinieren kann. Durch das Hinzufügen von realitätsnahen Visualisierungsmethoden in eine wissenschaftliche Darstellung, läßt sich eine zusätzliche Qualität und Aussagekraft der erzeugten Bilder und Sequenzen schaffen (Abbildung 7.77 und 7.78).

Wie dort im Beispiel zu sehen, erschließt sich dem Betrachter schnell der für ihn interessante Bereich - hier Feuer und Rauch. Auf einen Blick kann er aufgrund der zusätzlich eingebrachten Visualisierungsmethode für Strömungslinien schnell den Verlauf der Strömung und damit den Verlauf des Rauches und der Brandgase erkennen. Die gesamte Darstellung wird einfacher und direkter interpretierbar. Diese kombinierte Methode ist vor allen Dingen für die Präsentation gegenüber Laien und Fachfremden interessant, da man ihnen auf leicht zu verstehende Art und Weise die wichtigen Eigenschaften des untersuchten Datensatzes vermitteln kann. Um diese Vorteile mit in ein System zur Visualisierung von Strömungssimulationsdaten zu übernehmen, ergibt sich also die Anforderung, dass man in diesem System beliebig technisch-wissenschaftliche und realistische Visualisierungsmethoden mischen können sollte.

Kombiniert man diese Darstellung noch mit der Forderung nach realistischer Repräsentation der Umgebungsgeometrie aus dem vorangegangenen Kapitel, so verbessert das noch einmal die Benutzer-Orientierung in der Szene und damit innerhalb des untersuchten Datensatzes. In vielen Präsentationen bedient man sich mittlerweile dieses Stilmittels: Es werden Visualisierungsmethoden mit realistischer Darstellung gemischt, um dem Betrachter einen schnellen Einblick über die Problematik zu geben (Abbildung 7.77). Diese Vorgehensweise ist auch in anderen Visualisierungsbereichen schon eine Weile bekannt und üblich. So werden beispielsweise bei virtuellen Flügen durch das Sonnensystem, neben „echt“ wirkenden Planeten, gerne Umlaufbahnen als Ellipsen eingezeichnet, oder in der TV Branche werden errechnete Wetterdaten als realistisch wirkende Wolken zusammen mit Linien, die Windströmungen oder Hoch-/Tiefdruckgebiete abgrenzen, dargestellt (Abbildung 4.4).

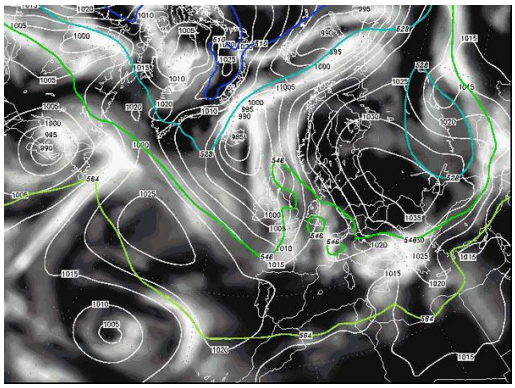


***Abbildung 4.4:*** *Wettervorhersage [wet05b]*

Auch hier geht man mittlerweile immer mehr dazu über, realistische Visualisierungen der Umgebung mit wissenschaftlichen Visualisierungsmethoden von Simulationsdaten am Rechner zu mischen.

### 4.1.5 Steuerung der Visualisierungsmethoden

Aufgrund der Menge an Simulationsdaten, die moderne Rechnerverbunde zu produzieren im Stande sind, wird es - wie beschrieben - für den Betrachter der Szene immer schwieriger, sich darin zurechtzufinden. Neben der Unterstützung der Navigation durch Einblendung der (realistisch wirkenden) Umgebung gibt es noch weitere Methoden, dem Nutzer die Darstellung der Steuerung in der Datenmasse zu erleichtern. Hierbei sind besonders drei Punkte von Bedeutung, die im Folgenden kurz erläutert werden.

#### 4.1.5.1 Räumliche Einschränkbarkeit

Was auf den ersten Blick trivial erscheint, ist für den Anwender von entscheidender Bedeutung. Man stelle sich eine Simulation vor, die das Weltklima mit einer sehr feinen Auflösung errechnet. Alle Daten der Simulation würden gespeichert und der Nutzer wäre imstande, sich diese Datenmenge komplett anzuschauen. Angenommen er hätte besonderes Interesse an den Luftdruckverhältnisse rund um den Äquator und den Windverhältnissen im Europäischen Raum. Sofort wird klar, dass ihm an dieser Stelle die Möglichkeit gegeben werden muss, die Visualisierungsmethoden örtlich zu begrenzen. So kann er eine Darstellung des Luftdruckes auf die Äquatorgegend beschränken und gleichzeitig die Windströmungsvisualisierung auf Europa. Würden beide Visualisierungsmethoden immer auf der gesamten Datenmenge aktiv - im Beispiel die komplette Erdkugel - hätte der Nutzer nur schwer die Möglichkeit (genau) abgegrenzte Bereiche zu untersuchen; von Leistungseinbußen einmal abgesehen, die die Berechnung und Darstellung der vielen „uninteressanten“ Daten ausserhalb des untersuchten Bereiches mit sich bringen würden.

Aus diesem Kontext erwächst die Anforderung, dass dem Nutzer die Möglichkeit gegeben werden sollte, den räumlichen und zeitlichen Bereich aus der Datenmenge, auf den er eine bestimmte Visualisierungsmethode anwenden möchte, wählen zu können.

#### 4.1.5.2 Mehrere gleichzeitig aktive Visualisierungsmethoden

Wie schon in Kapitel 4.1.4 angedeutet, ist es durchaus wünschenswert, mehrere Visualisierungsmethoden gleichzeitig auf die untersuchten Daten anzuwenden, beispielsweise wenn der Anwender zur gleichen Zeit eine Strömungslinien- und eine Partikel-Visualsierungsmethode auf den errechneten Strömungsdaten aktviert. Hierbei greifen beide Visualisierungsmethoden auf dasselbe Feld des Datensatzes zu: Die Windströmung.

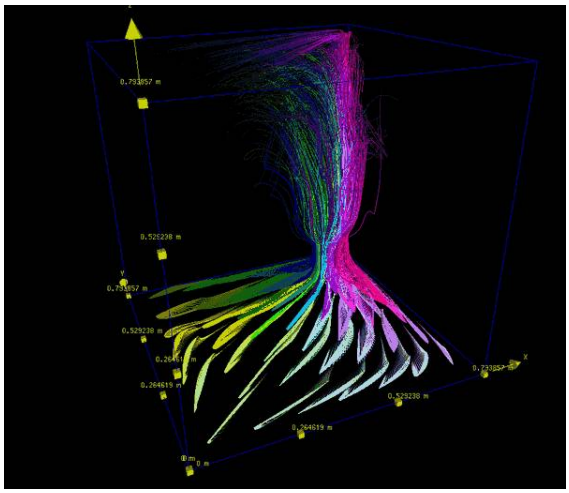
Da ein Datensatz jedoch, wie in Kapitel 3.5 beschrieben, aus mehreren verschiedenen Feldern bestehen kann (beispielsweise ein Temperatur-, ein Druck- und ein Strömungsfeld), so ist es auch möglich, dass die gleichzeitig aktiven Visualisierungsmethoden auf unterschiedliche Felder desselben Datensatzes zugreifen (Abbildung 4.1).

Als Anforderung aus diesem Sachverhalt läßt sich definieren, dass eine Visualisierungsumgebung in der Lage sein muss, zum einen mehrere gleichzeitig aktive Visualisierungsmethoden zu unterstützen. Zum anderen muss es möglich sein, dass diese Visualisierungsmethoden gleichzeitig auf verschiedene Felder des untersuchten Datensatzes zugreifen können.

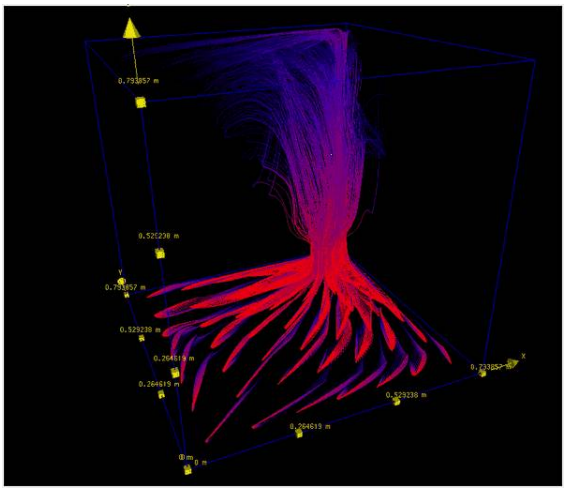
#### 4.1.5.3 Interaktive Steuerung der Parametersets

Jede Visualisierungsmethode bringt eine andere Art und Weise mit sich, mit Hilfe von mathematischen Methoden und Algorithmen, eine Darstellung aus den zugrundeliegenden Strömungsdaten zu produzieren. Die meisten dieser Methoden sind parametrisiert, d.h. man kann ihr Ergebnis und damit im Endeffekt die von ihnen errechnete Darstellung mit Hilfe von Parametern steuern. Die meisten der Visualisierungsmethoden beruhen auf numerischen Lösungsverfahren. Diese Verfahren können beispielsweise gegen präzisere ausgetauscht werden (z.B. einen Euler Integrator gegen ein Runge Kutta Verfahren höherer Ordnung). Zusätzlich können auch Dinge wie Toleranz, Fehlerschwellwerte oder maximal zulässige Schrittweiten festgelegt werden.

Neben den rein mathematischen Steuerparametern der Visualisierungsmethoden gibt es auch eine Reihe von Parametern, die direkten Einfluß auf die optische Erscheinung der Visualisierung haben. So kann man beispielsweise eine Strömungslinie anhand ihrer Länge oder ihres Ursprunges auf einer Fläche verschieden einfärben. Die Farbverteilung hierbei kann gesteuert werden (Abbildung 4.5).



Streamlines gefärbt anhand der Startposition (Regenbogen)



Streamlines gefärbt anhand ihrer Länge (von Rot nach Blau)

***Abbildung 4.5:*** *Unterschiedliche Parametersets für dieselbe Visualisierungsmethode.*

All diese Parameteränderungen sollten zur Laufzeit, d.h. bei aktiver Darstellung der Visualisierungsmethode, änderbar sein, um den direkten Einfluß der Parameteränderung auf die Darstellung untersuchen zu können.

Die zugehörige Anforderung, die sich hieraus kondensieren läßt, besagt, dass es in einem Visualisierungssystem möglich sein sollte, die Steuerparameter jeder aktiven Visualisierungsmethode interaktiv manipulieren zu können.

## 4.2 Technische Anforderungen

Neben den rein konzeptionellen Anforderungen an ein Visualisierungssystem, lassen sich darüber hinaus aus verschiedenen Rahmenbedingungen und Problemen auch rein technische Anforderungen definieren. Hier steht das System als solches im Vordergrund und die damit zusammenhängenden einzelnen technischen Fähigkeiten. Die Umsetzung dieser technischen Anforderungen bildet eine wichtige Grundlage für ein funktionelles und effizientes Visualisierungssystem. Wird sie realisiert, so ermöglicht sie dem Nutzer eine Reihe von Vorteilen, von denen er sehr stark profitieren kann. Er erhält ein hochwertiges Werkzeug zur Darstellung von Strömungsdaten. Diese technischen Anforderungen werden im Folgenden vorgestellt.

### 4.2.1 Unterstützung großer Datenmengen / Geschwindigkeit der Darstellung

Wie bereits in Kapitel 4.1.1 deutlich gemacht, fallen bei der Strömungssimulation typischerweise schnell große Datenmengen an, die es in der anschließenden Anwendung von Visualisierungsmethoden zu bewältigen gilt. Die konzeptionelle Anforderung, die hieraus erwächst, läßt sich unter dem Titel „Kompression“ zusammenfassen. Diese Fähigkeit hat auch eine technische Seite. So muss z.B. auch technisch sichergestellt sein, dass die verwendete Plattform fähig ist, die genannten Daten zu speichern und adressieren zu können. Dabei spielt neben der reinen Speichergröße, die man mit Kompression in den Griff bekommen kann, unter anderem auch die Zugriffszeit eine entscheidende Rolle. Hier wird es wichtig, dass man in erster Linie sehr schnell Zugriff auf dedizierte Bereiche in der Datenmenge erhält. Der intelligente Aufbau des Datenformates ist dafür von entscheidender Bedeutung. Außerdem muss es möglich sein, wie in Kapitel 4.1.5.2 beschrieben, eben nur die relevanten Bereiche in ausreichender Auflösung aus dem Datensatz lesen zu können. Fatal wäre es, wenn statt dessen bei jedem Zugriff immer nur der komplette Datenbestand geladen werden kann. Es müsste anschließend aufwendig erst im Arbeitsspeicher die notwenige Filterung vornommen werden. Eine schnelle und intelligente Datenstruktur und Adressierung ist hier von entscheidender Bedeutung und beschreibt eine wichtige technische Anforderung an das Gesamtsystem.

### 4.2.2 Modularisierung

Wird eine Visualisierungsumgebung modular aufbaut, kann man daraus mehrere Vorteile ziehen. Einer der wichtigsten ist es, dass die einzelnen Komponenten getrennt und weitestgehend unabhängig voneinander entwickelt werden können. Wichtige Module, die hierbei entstehen können, sind z.B. ein Modul für die grafische Benutzeroberfläche, eines für die Graphikausgabe, eines für die Steuerung, eines für die Datenhaltung bzw. den Im- und Export oder für die Visualisierungsmethoden (Berechungen) und deren Verwaltung. Werden diese Module über abstrakte Schnittstellen und ein Nachrichtensystem miteinander verbunden, so können sie im asynchronen Betrieb nebeneinander parallel ablaufen. Hier gibt es bereits einige Visualisierungssysteme, die dieses Prinzip verwenden (z.B. Covise [COV]). So wird es möglich, im Bedarfsfall einzelne Komponenten flexibel gegen andere austauschen zu können.

So kann man beispielsweise die Benutzeroberfläche gegen eine andere austauschen, oder ein bestimmtes Ausgabeformat (Szenegraph) gegen ein anderes auswechseln. Neben dem Vorteil, dass die unerschiedlichen Bereiche getrennt voneinander angepaßt und optimiert werden können, steigert diese Vorgehensweise also die parallele Effizienz und die Wiederverwertbarkeit der einzelne Systembestandteile und damit des Gesamtsystems. Eine schnelle und indivuelle situationsgebundene Anpassbarkeit wird gewährleistet (vergleiche Abbildung 4.6).

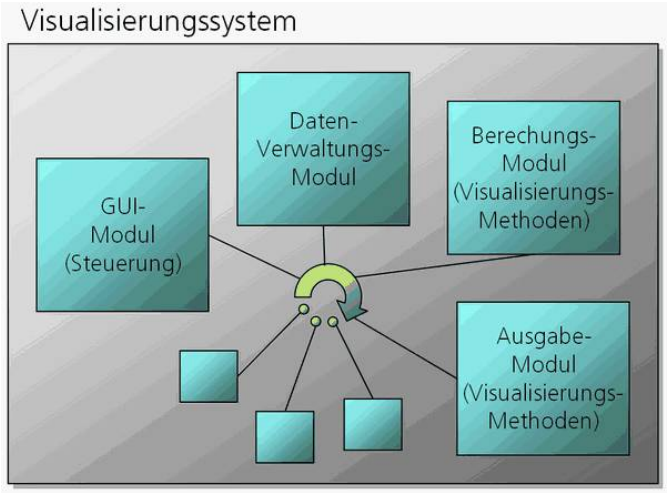



***Abbildung 4.6:*** *Beispiel für einen modularen Aufbau: Einzelne Funktionsbereiche des Visualisierungssystems befinden sich in getrennten Modulen.*

### 4.2.3 Schnelle Anpaßbarkeit

Um sich an wechselnde Randbedingungen anzupassen, ist es wichtig, dass man einzelne Komponenten des Systems leicht erweitern bzw. ergänzen oder auch austauschen bzw. komplett weglassen kann. So muss es z.B. möglich sein, neben der Integration neuer Visualisierungsmethoden und -verfahren, auch mit anderen Systemkomponenten ähnlich zu verfahren.

Angenommen es wäre nötig, in ein bestehendes Visualisierungssystem für Geometrie mit bereits integriertem Benutzerinterface und / oder einem bestimmten Ziel-Szenegraph der Ausgabe lediglich die Darstellungsfähigkeit für Strömungsdaten zu integrieren, so wäre es wünschenswert, wenn man aus dem modularen Aufbau des hier entwickelten Systems lediglich die betroffenen Teile herauslösen kann (im Beispiel die Datenhaltung und die Berechnung der Visualisierungsmethoden), um sie in das Zielsystem zu integrieren. Die graphische Darstellung müsste dann nur an den vorgegebenen Szenegraphen und die Parametersteuerung der Visualisierungsmethoden das bereits vorhandene Benutzerinterface angepaßt werden (siehe Abbildung 4.7).

Ein anderes Anwendungsbeispiel ist die Entwicklung eines serverbasierten Berechnungsmoduls für die Geometrie der Visualisierungsmethoden. Die Benutzeroberfläche wird durch eine Steuerung über eine Netzwerkanbindung ersetzt. Die von den Visualisierungsmethoden errechneten Geometriedaten werden, anstatt sie direkt über einen Szenegraph auszugeben,

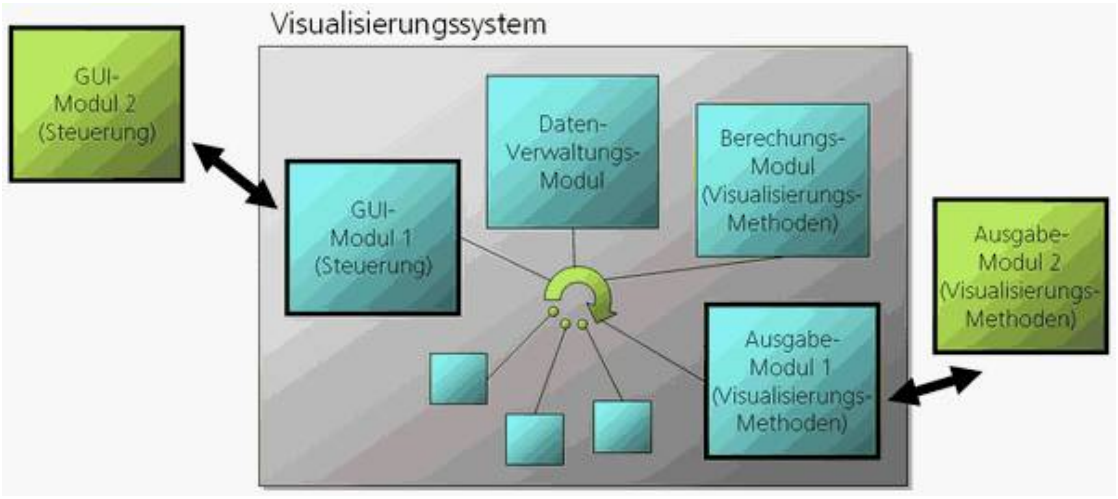



***Abbildung 4.7:*** *Einzelne Module werden durch andere ersetzt, um das System an andere Randbedingungen anzupassen (z.B. Wechsel der GUI- oder Graphiklibrary).*

über das Netzwerk zurück zum Empfänger übertragen (siehe Abbildung 4.8). In diesem Beispiel werden also nicht nur einzelne Module ersetzt, sondern manche komplett weggelassen.

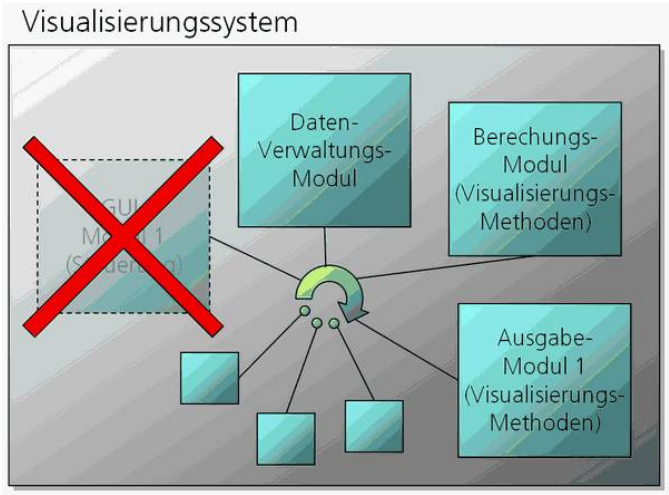



***Abbildung 4.8:*** *Einzelne Module werden entfernt, da sie nicht benötigt werden. Beispielsweise wird die GUI Steuerung redundant, da die Steuerdaten über ein Netzwerk empfangen werden.*

Dadurch, dass die verwendeten Module, wie in Kapitel 4.2.2 beschrieben, über abstrakte Schnittstellen per Nachrichtenversand kommunizieren sollen, besteht die Möglichkeit, Teile des Systems komponentenweise wiederzuverwenden. Man muß lediglich die benötigten Module zusammenziehen und an neues Problem anpassen. So kann man die Visualisierung schnell an unterschiedliche Randbedingungen anpassen.

### 4.2.4 Schnelle Erweiterbarkeit

Für ein Visualisierungssystem von Strömungsdaten ist es von entscheidender Bedeutung, dass es eine Vielzahl von unterschiedlichen Visualisierungsmethoden unterstützt bzw. mit sich bringt - sowohl für Skalarfelder als auch für Vektorfelder. Wie bereits in vorherigen Kapiteln verdeutlicht, extieren bereits eine Menge etablierter Visualisierungsmethoden für diese Feldtypen. Trotzdem macht der Fortschritt auf vor diesem Bereich nicht halt und so wurden auch hier in den letzten Jahren immer wieder neue Methoden und Verfahren veröffentlicht, die vor allen Dingen durch die Weiterentwicklung der Hardware [Höh02] möglich werden. Dabei werden zum einen bekannte Methoden auf neuem Wege realisiert, die meist durch die immer weiter steigende Leistungsfähigkeit der Hardware erst möglich geworden sind. Zum anderen entstehen auch komplett neue Herangehensweisen, die es vorher noch in keiner Form gab.

An dieser Stelle wird deutlich, wie wichtig es für das Visualisierungssystem ist, solche Änderungen bzw. Erweiterungen hinsichtlich neuer und verbesserter Visualisierungsmethoden und Verfahren schnell aufnehmen und umsetzen zu können. Dieser Punkt stellt eine weitere technische Anforderung dar. Durch eine geeigente Programmstruktur und Methodenverwaltung muss es möglich sein, jederzeit neue Visualisierungsmethoden, die z.B. auf komplett neuen Algorithmen oder auf an spezielle Hardware angepaßten Verfahren beruhen, in das System zu integrieren und es damit aktuell zu halten. So hat man jederzeit die Möglichkeit, die im System vorhandenen Darstellungsvarianten an bestimmte Bedingungen und Wünsche seitens des Anwenders anzupassen, was uns zum nächsten Unterkapitel führt.

### 4.2.5 Betriebssystem unabhängig

Visualisierungssysteme für Strömungsdaten, die heutzutage auf dem Software Markt verfügbar sind, sind in der Regel an ein bestimmtes Betriebssystem angepaßt. Prominente Beispiele sind hier beispielsweise Covise [COV] für Linux oder FemLAB [FEM] für Windows. Kaum ein System auf dem Markt bietet die Möglichkeit, bei der Entwicklung oder Anpassung das Betriebssystem zu wechseln.

Um den Wechsel von einer Betriebssystemumgebung in eine andere zu ermöglichen, ist es wichtig, dass die verwendeten Programmierbibliotheken portabel sind. Spezialisierte Lösungen sollten vermieden werden. So ist es beispielsweise bei der Wahl der Programmiersprache, der GUI und Graphikbibliotheken oder dem Ausgabe-Szenegraph wichtig, dass sie Plattform unabhängig sind. Dadurch besteht dann die Möglichkeit, das System beispielsweise sowohl auf einem Linux Cluster als auch auf einem MAC oder einer Windows Workstation zu betreiben.

### 4.2.6 GUI unabhängig

Heutzutage existiert eine Vielzahl von unterschiedlichen Bibliotheken auf dem Softwaremarkt, mit deren Hilfe man *graphische Benutzeroberflächen (GUI)* zusammenstellen kann.

Diese Bibliotheken enthalten Elemente, wie beispielsweise Eingabezeilen, Buttons, Scrollbalken oder Menüs. Bekannte Beispiele hierzu sind die Microsoft Foundation Classes [MFC] oder QT [QT].

Möchte man nun die Möglichkeit gewährleisten, dass man einzelne Systemkomponenten wiederverwenden kann, so muss man dafür sorgen, dass man die Teile des Systems, die keinen direkten Kontakt mit der GUI haben, wie beschrieben, weitestgehend von dieser abkapselt. Dadurch garantiert man zum einen, wie in Kapitel 4.2.5 beschrieben, das man Teile des Systems komplett ohne GUI betreiben kann, andererseits wird es allerdings auch ermöglicht, die GUI Komponente gegen eine andere auszutauschen. Optimal ist der Fall, in dem die GUI Bibliothek selbst Betriebssystem unabhängige Objekte zur Verfügung stellt.

### 4.2.7 Hardware unabhängig

Wie Kapitel 4.2.5 beschreibt, ist eine wichtige technische Anforderung, die Betriebssystem Unabhängigkeit. So ist man in der Lage, das Visualisierungssystem auf mehreren Plattformen zu betreiben. Wenn man über diese Softwareebene hinaus auf die Hardware schaut, kommt schnell ein anderer wesentlicher Punkt zum Vorschein: Die Hardware. Manche Visualisierungssysteme setzten neben bestimmten Betriebssystemen auch das Vorhandensein bestimmter Hardwarekomponenten voraus, sind darauf zum Teil sogar optimiert und damit nur auf diesen lauffähig.

Durch Standardisierungen im Graphik Bereich (OpenGL [SGIc], DirectX [Mica]) wurden hier in den letzten Jahren deutliche Verbesserungen erreicht. So muss meist nur noch dafür Sorge getragen werden, dass die entsprechende Graphikbibliothek auf dem jeweiligen System vorhanden ist, um eine Portierung zu ermöglichen. Durch eine Reihe von Entwicklungen in letzter Zeit wurde dieser Idee jedoch mit der Einführung von speziellen Graphik Extensions und Technicken wie „Shadern“ (siehe Kapitel 3.7) gewissermaßen entgegengesteuert. Durch Ausnutzung spezieller Hardware Gegebenheiten wurden viele hochperformante Einzellösungen möglich.

An dieser Stelle ist ein Konzept gefragt, das es dem Nutzer erlaubt, zum einen Plattform unabhängige Visualisierung zu betreiben, auf der anderen Seite aber die Möglichkeiten dedizierter Hardware angepasster Optimierungen nicht verschließt. So kann die Visualisierung, sofern entsprechende spezialisierte Hardware zur Verfügung steht, davon auch profitieren, d.h. an sie angepaßte und beschleunigte Algorithmen verwenden.

Abseits von der GPU ist es auf Seiten des Prozessors seit längerem kein allzugroßes Problem mehr, durch Verwendung von höheren Programmiersprachen wie beispielsweise C++, von der Hardware weitestgehend abgekoppelt und damit in diesem Sinne Plattfrom unabhängig zu entwickeln. Hier ist also lediglich zu beachten, dass man sich einer solchen Programmiersprache und entsprechender Hilfsbibliotheken bedient, um diesen Vorteil ausnutzen zu können.

### 4.2.8 Szenegraph unabhängig

Hand in Hand mit der Forderung der Unabhängigkeit von spezieller Graphik-Hardware geht die Anforderung, von darauf aufsetzenden speziellen Szenegraphen unabhängig zu sein. Ähnlich der angestrebten Unabhängigkeit von der verwendeten GUI Bibliothek (siehe Kapitel 4.2.6), dient diese Forderung der Sicherstellung einer schnellen und flexiblen Anpaßbarkeit des Gesamtsystems im Bedarfsfall. Wird das Einsatzgebiet der Applikation bzw. der zu verwendende Output Szenegraph gewechselt (beispielsweise durch den Wechsel der Plattform oder der umhüllenden Visualisierungsumgebung), so muss es auf einfache Art und Weise möglich sein, die entsprechenden Systemkomponenten, die an spezifische Szenegrafen angepaßt sind, durch andere auszutauschen. So müssen nur die betroffenen Module gewechselt / angepaßt werden, der Rest des Systemes bleibt unverändert. Auf diese Art und Weise wird der angestrebte schnelle Wechsel zwischen verschiedenen Szenegraphen unterstützt.

### 4.2.9 Datenformat unabhängig

Strömungsdaten, die mit entsprechender Simulationssoftware errechnet werden, finden sich heutzutage, wie bereits in Kapitel 3.3, beschrieben in einer Vielzahl verschiedener Speicherformate wieder. Allen diesen Speicherformaten gemeinsam ist der Umstand, dass sie in erster Linie darauf ausgelegt sind, die Daten schnell und effizient abspeichern zu können. Hierbei wird der Kompression (vgl. Kapitel 4.1.1) oder der Einlese-Geschwindigkeit wenig Aufmerksamkeit gewidmet. Um nun eine Visualisierungsumgebung mit einer Unterstützung für diese Datensätze auszustatten, müssen in erster Linie viele verschiedene Import Funktionen implementiert werden. Für jedes der gewünschten Formate ein eigenes. Nach dem Einlesen der Daten müssen diese allerdings noch sinnvoll interpretiert werden. Da bei einer Strömungssimulation typischerweise immer mehrere Felder pro Datensatz abgelegt werden (beispielsweise Temperatur, Luftdruck, Luftströmung, usw.) bleibt es anschließend der Visualisierungsanwendung überlassen, die gewünschten Werte aus der Fülle der in den jeweiligen Dateien abgelegten Feldern auszulesen. Augrund der vielen verschiedenen Zelltypen und Gitterstrukturen kann es mitunter sehr aufwendig werden, für alle entstehenden Möglichkeiten entsprechende Einlese und Interpolationsroutinen bereitzuhalten (vgl. Kapitel 3.3.1).

Um von den zahlreichen Input Formaten unabhängig zu werden und gleichzeitig der in Kapitel 4.1.1 geforderten Kompression gerecht zu werden, zusammen mit der Unterstützung großer Datenmengen (Kapitel 4.2.1), ist die Entwicklung eines eigenen unabhängigen Formates eine weitere technische Anforderung, die an dieser Stelle definiert wird. Auf diese Weise ist man nicht an die Randbedingungen der Eingabeformate gebunden, sondern kann darüber hinaus neue Techniken und Vorgehensweisen entwicklen.

### 4.2.10 Leistungs / Speichergrößen unabhängig

Wie bereits erläutert, resultieren Simulationen im Bereich der Computational Fluid Dynamics oft in großen Datenmengen, verursacht durch sehr feine räumliche oder zeitliche Diskretisierung des Simualtionsraumes bzw. der Simulationszeit. Auf weniger leistungsfähigerer

Hardware stößt man hier schnell an natürliche Grenzen: Der Arbeitsspeicher ist zu klein oder zu langsam, die CPU ist nicht schnell genug, um alle gewünschten Operationen auf dem gesamten Datenraum auszuführen, die für eine Visualisierung der Daten notwendig sind. Mit der Kompression der Daten, die es ermöglicht, sie dichter in den Speicher zu packen oder des Daten Clippings, hat man nun die Möglichkeit eine weitere technische Anforderung zu formulieren: Die Leistungs- und Speichergrößen Unabhängigkeit des Visualisierungssystems. Wird sie umgesetzt, so kann man auch auf weniger leistungsfähigen Geräten mit durchschnittlicher Speicherausstattung, die Strömungsdaten untersuchen. An dieser Stelle ist das Problem zu lösen, nach welcher Methodik entsprechende Teilbereiche aus der Menge der zur Verfügung stehenden Daten, gefiltert und entsprechend reduziert werden müssen.

## 4.3 Zusammenfassung

Zusammenfassend läßt sich sagen, dass viele unterschiedliche konzeptionelle und technische Anforderungen an ein Visualisierungswerkzeug für wissenschaftliche Strömungs-Simulationsdaten existieren:

- Aufgrund der zu bewältigenden Datenmenge muß die Visualisierung parallelisiert werden, um dem Benutzer dennoch Interaktivität mit den Daten zu erlauben.
- Die Visualisierung muß unabhängig von den Inputdatenformaten funktionieren. Nach einem Konvertierungsschritt sollte ein für die reine Ausgabe optimiertes Datenformat verwendet werden, welches sowohl Parallelisierung als auch Kompression, Progressivität und damit das Nachladen von Details in interessanten Datenteilbereichen unterstützt.
- Das Visualisierungssystem muß voll skalierbar sein, um sich optimal an die jeweils gegebene Hardwareplattform anpassen zu können.
- Das Visualisierungswerkzeug muß so entworfen werden, dass es leicht um zusätzliche neue Visualisierungsmethoden erweitert werden kann, die sich sowohl auf klassische Methoden als auch auf hardwareoptimierte neue Algorithmen stützen können sollten.
- Durch einen Plattform unabhängigen modularen Aufbau wird garantiert, dass kritische Komponenten und / oder Teilmodule schnell ergänzt oder gegen andere, an den Einzelfall speziell angepaßte, ausgetauscht werden können.
- Es muß möglich sein, die zugrundeliegende Geometrie mit in die Analyseszene einzublenden. Die Einblendung einer realistisch wirkenden Geometrie hilft sowohl dem Wissenschaftler als auch dem unspezialisierten Anwender, die Ergebnisdaten sinnvoll einzuordnen. Zusammen mit der Mischung von technisch-wissenschaftlichen und realitätsnahen Visualisierungsmethoden wird dadurch eine neue Qualität der Darstellung erreicht.

Es gilt also, ein Konzept zu entwerfen, welches sowohl auf die Möglichkeiten im Bereich der optimierten und parallelisierten Ausgabe seitens der Hardware eingehen kann (spezielle Graphikbefehle aktueller Graphikchips), als auch gleichzeitig die dafür notwendigen Vorberechnungen (durch Parallelisierung) beschleunigt. Zusätzlich muß dieses System auch in Zukunft leicht erweiterbar sein, um flexibel auf die jeweils aktuellen Änderungen im schnell lebigen Graphik-Hardware-Markt eingehen zu können. Das Ziel ist ein universelles und modularisiertes Visualisierungswerkzeug für Strömungsdaten.

# Kapitel 5

# Konzept

In diesem Abschnitt wird basierend auf den in Kapitel 3 vorgestellten Grundlagen und unter Vorraussetzung der in Kapitel 4 aufgestellten Anforderungen ein Konzept für ein massiv paralleles Visualisierungssystem für Strömungsdaten entwickelt. Hierdurch wird eine Lösung des in Kapitel 1 präsentierten Problems erabeitet. Die einzelnen Anforderungen werden aufgegriffen und in verschiedenen Konzeptteilen umgesetzt. Die Beschreibung dieser Teile zerfällt in mehrere Unterkapitel, die im Anschluß aufgelistet sind.

## 5.1 Übersicht

Im Gegensatz zum Gebiet der Strömungs-Simulation, auf dem durch parallele Programmierung in der Vergangenheit bereits eine Menge erreicht wurde, hat die Visualisierungsseite in dieser Richtung noch viel Entwicklungspotential. Schon vor einiger Zeit hat man auf der Simulationsseite erkannt, dass man durch parallelisierten Programmcode erhebliche Leistungssteigerungen erreicht. Heute auf dem Softwaremarkt erhältliche Strömungs-Simulations-Softwarepakete sind bereits mit Mehrprozessor- und/oder Cluster-Unterstützung verfügbar [FLU]. Durch die Übertragung des parallelen Programmierkonzeptes von der Simulation auf die zugehörige Visualisierungsseite ist ein Performancegewinn zu erwarten, der ähnlich groß sein kann. Auf diese Weise wird es möglich, die vorhandenen Leistungsunterschiede zwischen physikalischer Simulationsrechnung und deren optischer Ausgabe zu schmälern, bzw. ganz zu beseitigen. Parallele Visualisierung als solche ist kein neues Thema. Sie wurde bereits erfolgreich in vielen Fällen eingesetzt (vgl. Kapitel 2). Sie jetzt auf Strömungs-Simulationsdaten anzuwenden, ermöglicht auch hier die Vorteile von mehreren Prozessoren und Rechnern zur Geschwindigkeitssteigerung auszunutzen.

Im Gegensatz zur Simulationsseite ist es jedoch ungleich schwieriger, das Cluster Parallelisierungskonzept auf die Visualisiersmethoden zu übertragen, da die nötigen Berechnungen für die Darstellung der Daten eher selten aus wenigen aufwendigen, als vielmehr aus sehr vielen relativ einfachen Kalkulationen bestehen. Durch den Overhead und die Latenz, die zwangsläufig durch die nötige Synchronisationskommunikation (z.B. Netzwerkkommunikation in einem Cluster) hinzukommt, wird die Performance, die man durch die Parallelisierung

hinzugewinnt, aufgrund der benötigten Menge von Synchronisierungsvorgängen wieder gemindert. Allerdings gibt es auch auf diesem Gebiet bereits vielversprechende Entwicklungen, die sowohl auf der Softwareseite (WireGL [WIR]), als auch direkt auf der Hardware-Output-Seite ansetzten (Lightning2 [LIG]). Software Lösungen sind hierbei in der Regel flexibler und lassen sich leicht auf andere Systeme übertragen. Hardware Lösungen setzen aufgrund ihrer Natur immer spezielle Hardware Gegebenheiten voraus. So wird es z.B. schwierig, das System durch Austausch einzelner Komponenten auf dem neuesten Stand zu halten, da diese u.U. dann nicht mehr kompatibel zum Rest sind. Außerdem sind hier derzeit nur sehr wenige und sehr teure Alternativen auf dem Markt verfügbar. Auch aus diesem Grund wird in dieser Arbeit die Software Seite zur Parallelisierung gewählt.

Heutzutage sind bereits eine Vielzahl wissenschaftlicher Visualisierungsmethoden bekannt (vgl. Kapitel 3.6.1), denen jedoch meist die oben beschriebene Optimierung durch Parallelisierung fehlt. Aus diesem Grund wird sich diese Arbeit auch intensiver mit der Parallelisierung einiger ausgewählter Visualisierungsmethoden befassen. Am Ende dieses Kapitels befinden sich dazu entsprechende Beispiele, die im Rahmen dieser Arbeit umgesetzt wurden.

Ausgehend von einzelnen Bausteinen und Modulen, deren Konzepte im Anschluß vorgestellt werden, wird dann in Kapitel 5.7 eine Gesamtsystem-Architektur entwickelt und schematisch präsentiert. Hier kann man dann noch einmal sehen, wie die einzelnen Komponenten zusammenhängen und miteinander zur Laufzeit kommunizieren.

## 5.2 Darstellung

In diesem Abschnitt werden die grundlegenden Konzepte für die Darstellung vorgestellt. Es werden verschiedene Methoden und Ansätze präsentiert, wie sich die technisch-wissenschaftliche Visualisierung von Strömungsdaten in ein Visualisierungssystem integrieren läßt.

### 5.2.1 Proben Konzept

Wie bereits in Kapitel 3.1 erläutert, bedient man sich zur Umwandlung der reinen Simulationsdaten in interpretierbare grafische Darstellung der Visualisierungsmethoden. Um nun die im vorangegangen Kapitel 4.1.5.1 beschriebene Anforderung zu erfüllen, Visualisierungsmethoden sinnvoll auf bestimmte Bereiche im VR-Raum einschränken zu können, wird an dieser Stelle das sogenannte *Probenkonzept* [SMS03b, SMS03b] eingeführt.

**Definition 5.1 (Probe)** *Eine Probe bezeichnet einen Teil des Datenraumes, meist quaderförmig. Dieser Teil kann beliebig im VR Raum platziert und in seiner Größe, Lage und Orientierung verändert werden. An diesen selektierten Bereich wird eine Visualisierungsmethode geknüpft. Die grundsätzliche Idee der Probe ist nun, dass nur in dem Bereich, der durch die Probe aus dem Datenraum herausgeschnitten wird, die gewünschte Visualisierungsmethode aktiv ist. Die Probe selbst kann dabei jede beliebige Form haben (Quader, Quadrat, Linie, Kugel, ...).*

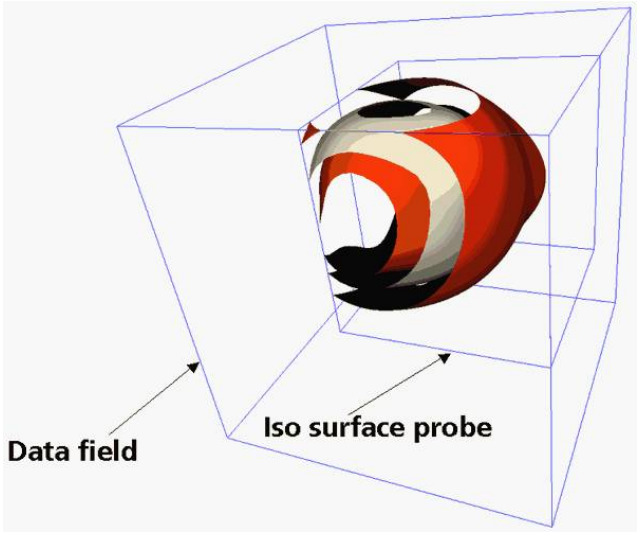



***Abbildung 5.1:*** *Probenkonzept: Eine Visualisierungsmethode wird nur innerhalb einer Probe (hier: der kleinere Würfel) berechnet. Die eigentliche Datenmenge ist größer (hier: der äußere Würfel).*

Diese Vorgehensweise bietet die Möglichkeit, dass man auf einfache Art und Weise die Darstellungen der Visualisierungsmethoden auf dedizierte räumliche Gebiete beschränken kann. Die Probe selbst kann beliebig positioniert und in ihrer Größe skaliert werden. Die entsprechenden Parameter für Platzierung und Skalierung werden in der Probe selbst gespeichert. Mit diesem Ansatz ist es somit auch möglich, die gesamte Datenmenge (und nicht etwa nur einen Ausschnitt davon) mit der Visualisierungsmethode der Probe darzustellen: Man vergrößert die Probe so lange, bis sie den gesamten Datenraum umfasst.

Ein weiterer wichtiger Vorteil dieses Konzeptes liegt darin begründet, dass man leicht verschiedene Visualisierungsmethoden mischen und gleichzeitig darstellen kann: Man aktiviert mehrere Proben mit unterschiedlichen Visualsierungsmethoden simultan und schiebt diese im VR Raum ineinander. So sind an den Überlappungsstellen alle Visualisierungsmethoden gleichzeitig zu sehen. Verdeckungen kann man durch Transparenzeinstellungen der betroffenen Visualisierungsmethoden gerecht werden. An den Stellen, wo nur eine der Proben aktiv ist, wird nur die jeweils von ihr definierte Darstellung wirksam. Ein schematischer Aufbau einer Probe ist in Abbildung 5.2 zu sehen.

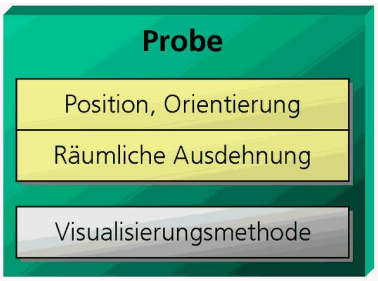



***Abbildung 5.2:*** *Aufbau einer Probe: Neben der Beschreibung ihrer Position, Orientierung und Größe enthält sie eine gekapselte Visualisierungsmethode, die in dem Bereich, den die Probe aus dem Datenraum ausschneidet, aktiv ist.*

Mit Hilfe der Proben hat der Anwender die Möglichkeit, seine wissenschaftliche Darstellung der Daten und die damit einhergehende Analyse gezielt auf konkrete Bereiche innerhalb des Datenraumes einzugrenzen. Neben der reinen Raumbeschränkung bietet dieser Ansatz indirekt die Fähigkeit, die gesamte Visualisierung auf dem jeweiligen Endsystem zu beschleunigen, da „unnötige“ Teile nicht bei der Darstellung bzw. Berechnung der Visualisierungsmethode mit berücksichtigt werden müssen. Dieser Ansatz führt zu einem wichtigen Konzeptteil, der im folgenden Unterkapitel vorgestellt wird. Die Proben kaspeln quasi den Teil der Visualisierungsmethoden, den alle gemeinsam haben: Position, Größe, Lage und Sichtbarkeit.

#### 5.2.1.1 Daten Clipping

**Definition 5.2 (Geometrie Clipping)** *Mit Geometrie Clipping bezeichnet man den Vorgang, aus einer Menge von vorhandenen Geometriedaten bestimmte Bereiche „wegzuschneiden“. Die Gründe hierfür können verschieden sein. In der Regel wird dieses Verfahren kurz vor der Ausgabe der Daten angewendet, um den Aufwand für die Graphikhardware zu reduzieren. Hierbei werden bestimmte Bereiche, die vom Benutzer nicht einsehbar sind, vor der Darstellung als „außerhalb der Betrachterszene liegend“ erkannt und entsprechend weggeschnitten. Das prominenteste Beispiel für einen Geometrie Clipping Algorithmus stammt von Sutherland et al [SH74]. Es gibt auch noch andere Gründe, Geometriedaten zu clippen, beispielsweise um sich deren unnötige Übertragung über ein Netzwerk zu sparen [Sah03].*

Viele Strömungs-Visualisierungswerkzeuge, bedienen sich eines einfachen Tricks, um räumlich eingeschränkte Visualisierungsmethoden zu erzeugen: Sie berrechnen die Darstellung der gewünschten Visualisierungsmethoden auf dem gesamten Datenraum. Die dabei entstehende Geometrie, die dann zur Darstellung führt (beispielsweise Linien oder Pfeile bei einer Windströmung), wird erst nach diesem Berechnungs-Schritt entsprechend räumlich „beschnitten“, d.h. die uninteressanten Bereiche nachträglich ausgeblendet (siehe Abbildung 5.3). Hierbei ist eine Darstellung einer so räumlich eingeschränkten Visualisierungsmethode mitunter genauso zeitaufwendig oder sogar aufwendiger, wie die Darstellung der Visualisierungsmethode auf der kompletten Szene: Es werden (teuer) Geometriedaten errechnet, die nicht mit in die Darstellung einfließen. Das anschließende Beschneiden der entstandenen Visualisierungsmethoden-Geometrie kann mitunter sehr aufwendig werden, je nach entstandener Geometrie-Komplexität und Aufbau.


***Abbildung 5.3:*** *Übliche Reihenfolge: Zuerst werden die kompletten Daten an die Visualisierungsmethode übergeben, die daraus Geometrie errechnet. Erst unmittelbar vor der Ausgabe dieser Geometrie wird die darzustellende Datenmenge durch (Geometrie-)Clipping Verfahren reduziert.*

Der Nachteil dieser Vorgehensweise ist offensichtlich: Es werden zu viele Geometriedaten berechnet, auch auf „uninteressanten“ Gebieten. Hier bietet sich ein Rechenzeit-Einsparpotential, das einen Leistungszuwachs des Visualisierungssystems verspricht. Man kann bereits vor den Berechnungen der Visualisierungsmethode bestimmte Teile des Datenraumes ausblenden. Aktive und durch die Ränder einer Probe räumlich beschränkte Visualisierungsmethoden starten ihre Berechnungen, die sie zur Geometrieerzeugung benötigen, nur in den festgelegten Gebieten. Die Sichtbarkeitsgrenze wird durch den Rand der Probe bestimmt. Es sind nur Datenzugriffe auf die relevanten Bereiche nötig (siehe Abbildung 5.4).

**Definition 5.3 (Daten Clipping)** *Mit Daten Clipping wird in dieser Arbeit der Vorgang bezeichnet, schon vor dem Start der Berechnungen der Visualisierungsmethode, die für die Darstellung der Visualisierungsmethode irrelevanten Bereiche der Strömungsdaten „wegzuschneiden“ (da sie außerhalb der Proben liegen). Entsprechende Werte aus den so entfernten Teilen des Datenraumes müssen nicht in den Speicher eingelesen werden und fließen auch nicht in die Berechnung der Visualisierungsmethoden mit ein. Eine entsprechende Darstellungsgeometrie der geclippten Bereiche wird nicht erzeugt. Nur aus den verbleibenden Datenteilen wird die Geometrie der Visualisierungsmethode errechnet.*


***Abbildung 5.4:*** *Daten Clipping: Zuerst werden die Daten beschnitten. Nur noch die Teile, die relevant für den zu visualisierenden Datenraum-Ausschnitt sind, werden an die Visualisierungsmethode übertragen. Anschließend errechnet die Methode aus diesen Restdaten die zugehörige Geometerie für die Darstellung.*

Die Probe steuert also, welchen Teil aus dem Datenraum die eingeschlossene Visualisierungsmethode darstellen soll. Hierbei wird der Datenraum sowohl räumlich, als auch zeitlich eingeschränkt. Möchte man einen anderen Teil des Datenraumes visualisieren, so genügt es die, Probe an die entsprechenden Raum-/Zeit-Koordinaten zu verschieben.

### 5.2.2 Generisches Klassenkonzept für Visualisierungsmethoden

Um die Anforderung nach der schnellen Erweiterbarkeit zu erfüllen (vgl. Kapitel 4.2.3), wird ein Konzept benötigt, das es einfach macht, neue Visualisierungsmethoden in das System zu integrieren. Hier bietet sich ein generisches Klassenkonzept an. Will man neue Visualisierungsmethoden in das System integrieren, so läßt man sie von dieser Klasse erben und überlädt die dadurch vorgegebenen Funktionen.

In diese generische „Mutterklasse“ werden die wichtigsten Funktionen einer Visualisierungsmethode angelegt, die zur Integration in das Gesamtsystem benötigt werden. So wird sichergestellt, dass diese Funktionen in allen darauf aufsetzen Visualisierungsmethoden verfügbar

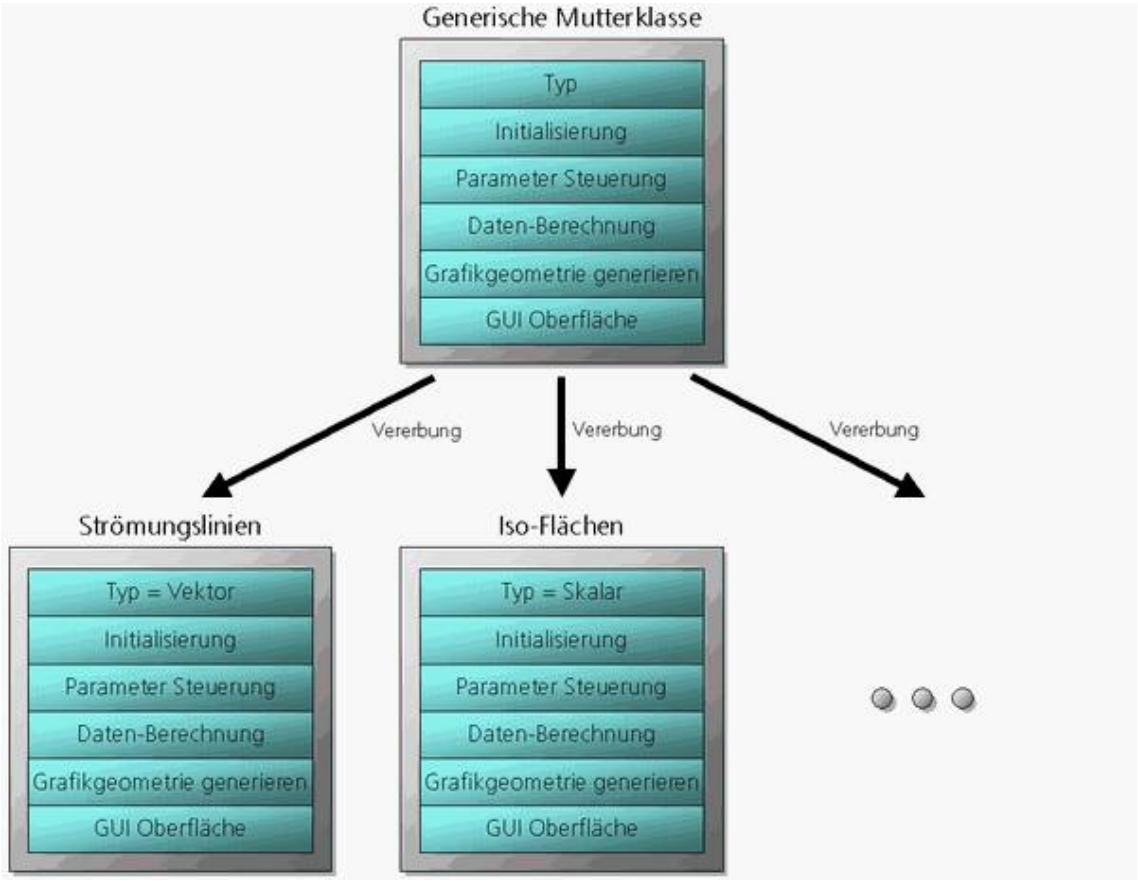



***Abbildung 5.5:*** *Generisches Klassenkonzept für Visualisierungsmethoden: Eine verallgemeinerte virtuelle Mutterklasse enthält alle wichtigen Funktionen. Die eigentlichen Ausprägungen der Visualisierungsmethoden erben von der Mutterklasse und überladen die dadurch festgelegten Funktionen.*

sind (siehe Abbildung 5.5) und trotzdem jede Visualisierungsmethode ihre eigene Ausprägung der jeweiligen Funktion haben kann.

Einige der wichtigsten Funkionen hierzu sind die folgenden:

**Funktionen zur Typabfrage**
: In dieser Funktion kann der „Typ“ der Visualisierungsmethode abgefragt werden. Typ bezeichnet hierbei, welche Art von Datenfeld von der Methode unterstützt wird (z.B. Skalar oder Vektor). Mit Hilfe dieser Information kann das System sicherstellen, dass nur Visualisierungsmethoden mit Datenfeldern verbunden werden, d.h. darauf angewendet werden können, die die richtigen Eingabedatenform beinhalten.

**Funktionen zur Initialisierung**
: Damit eine Visualisierungsmethode weiß, welche Daten aus welchem Datenfeld sie darstellen muss, dient diese Funktion zur entsprechenden Initialisierung. Neben der eigentlichen Verknüpfung mit dem entsprechenden Feld wird hier zusätzlich noch der Name und die Einheit der zu visulaisierenden physikalischen Größe an die Visualisierungsmethode mit übergeben (z.B. „Temperatur“ und „°C“ oder „Windgeschwindigkeit“ und

„km/h“).

**Funktionen zur Parametersteuerung**

Aufgrund der unterschiedlichen Natur jeder Visualisierungsmethode hat auch jede von ihnen andere Parameter, um ihre Erscheinungsform zu steuern. Beispielsweise kann man bei Iso Flächen den zugehörigen Iso Wert und eine Farbverteilung dazu festlegen. Bei Strömungslinien steuert man statt dessen den Ursprung der Linien, die Anzahl und die Länge. Hier gilt es pro Visualisierungsmethode unterschiedlich viele Parameter mit unterschiedlichen Typen zu beachten.

**Funktionen zur Berechnung**

Dieser Teil stellt den eigentlichen Kern der Visualisierungsmethode dar. Hier sind die größten Unterschiede zwischen den einzelnen Methoden zu finden. Verbunden mit einem Datensatz kann die Visualisierungsmethode in dieser Funktion beginnen, entsprechende Darstellungsgeometrien zu berechnen. Die Ergebnisse dieser Vorberechnungen werden in der Visualisierungsmethode gespeichert, damit anschließend aus ihnen Geometriedaten erzeugt werden können.

**Funktionen zur Generierung der grafischen Ausgabe**

In dieser Funktion werden aus den vorberechneten Daten die Darstellungsgeometrien erzeugt. Wird eine Visualisierungsmethode vom System dazu aufgefordert, einen Szenegraphknoten mit ihrer Geometrie zu generieren, so wird diese Methode aufgerufen. Als Ergebnis wird dann der entsprechende Knoten zurückgeliefert. Dieser Teil der Visualisierungsmethode wird über eine abstrakte Schnittstelle von der eigentlichen Implementierung der Visualisierungsmethode in ein extra Modul abgekoppelt, um im Bedarfsfall einen schnellen Wechsel auf ein andere Art der graphischen Ausgabe (z.B. Szenegraphwechsel) zu erlauben.

**Funktionen für die graphische Oberfläche**

In diesem Teil wird eine graphische Bedien-Oberfläche zur Visualisierungsmethode erzeugt. Mit ihrer Hilfe kann man dann die darin abgebildeten Parameter steuern. Es wird ein Fenster mit entsprechenden Steuerelementen (Buttons, Eingabefelder, Auswahllisten, etc.) bereitgestellt. Auch dieser Teil wird, wie die Generierung der grafischen Ausgabe zuvor, von der eigentlichen Implementierung der Visualisierungsmethode in ein extra Modul abgekoppelt, um auch hier einen schnellen Wechsel / Austausch im Bedarfsfall zu unterstützen.

Auf diese Weise wird es möglich, alle Visualisierungsmethoden mit denselben Kern-Funktionen zu versehen und die konkreten Ausprägungen der einzelnen Methoden zu verlagern. So hat das System eine genormte Vorgehensweise, um auf die einzelnen Methoden zugreifen zu können. Dieser Ansatz ist wichtig, um die Visualisierungsmethoden zur Laufzeit auf einfache Art und Weise gruppieren und zusammenfassen zu können, damit sie z.B. in einer Art Listenstruktur verwaltet werden können (siehe nächstes Unterkapitel).

#### 5.2.2.1 Visualization Method Pool

Wie in den Anforderungen (siehe Kapitel 4.1.2) definiert, ist die Unterstützung einer „multivariaten“, d.h. gleichzeitigen Darstellung mehrerer Visualisierungsmethoden ein wesentlicher Bestandteil des hier entwickelten Konzeptes. Durch den weiter vorne vorgestellten Proben-Ansatz, wird es möglich, mehrere Proben gleichzeitig zu aktivieren und gegebenenfalls sogar ineinander zu schieben. Um dieses Vorgehen zu unterstützten, wird an dieser Stelle ein weiterer wichtiger Konzeptteil entwickelt, die Idee des „Visualization Method Pools“:

Hier wird ein Ansatz benötigt, der es erlaubt, mehrere Proben (und die damit verknüpften Visualisierungsmethoden) simultan zur Laufzeit des Programmes zu aktivieren, diese zu verwalten, zu steuern und dabei parallel abzuarbeiten. Die Hauptaufgabe des restlichen Systems besteht dann nur noch darin, diese Menge an aktivieren Methoden bei der Darstellung zu durchlaufen und die entsprechenden Visualisierungen anzuwerfen. An dieser Stelle wird das Konzept des Visualization Method Pools eingeführt. Er besteht im wesentlichen aus einer dynamischen Menge, in die zur Laufzeit neue Proben eingefügt oder alte gelöscht werden können. In dieser Menge befinden sich alle zur Laufzeit aktiven Proben (siehe Abbildung 5.6).

**Definition 5.4 (Visualization Method Pool)** *Der Visualization Method Pool ist eine Speicherstruktur für Visualisierungsmethoden/Proben in Mengenform. Er kann zum einen dem Visualisierungssystem darüber Auskunft geben, in welchem Datenbereich welche Visualisierungsmethoden/Proben momentan aktiv sind. Zum anderen können über ihn diese aktiven Visualisierungsmethoden/Proben gesteuert und ggf. gelöscht werden. Wird vom System oder Nutzer eine neue Visualisierung aktiviert, so wird eine entsprechende Repräsentation in diesem Pool angelegt und gespeichert.*

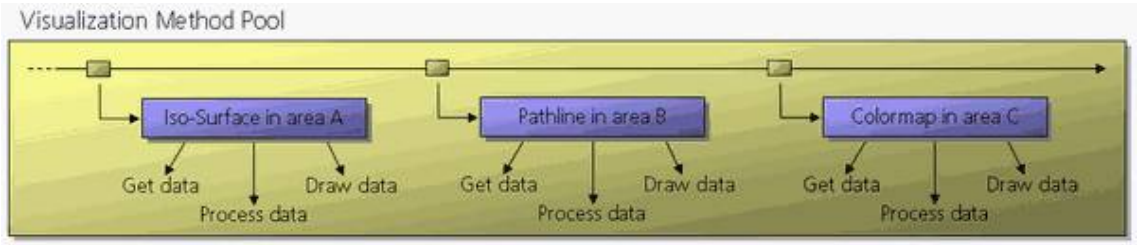



***Abbildung 5.6:*** *Visualization Method Pool: In einer dynamischen Mengenstruktur werden die zur Laufzeit aktiven Visualisierungsmethoden/Proben gespeichert.*

Um nun zu gewährleisten, dass alle vom User gewünschten Darstellungen pro Zeitschritt auch auf dem Bildschirm erscheinen, müssen vom System alle Elemente der Menge einmal bearbeitet werden. Erfahrungsgemäß wird diese Menge jedoch nie sehr viele Elemente enthalten, da zu viele verschiedene gleichzeitig aktive Visualisierungsmethoden nur noch schwer interpretierbare und „überladene“ Bilder liefern würden. Hier sind eher kleine Mengen zu erwarten, wenn auch der Pool beliebig groß werden kann (solange der Speicher dazu vorhanden ist).

## 5.3 Kommunikation

In diesem Abschnitt wird erläutert, wie sich die einzelnen Systemmodule untereinander verständigen bzw. Nachrichten austauschen. Die zentrale Instanz wird vorgestellt, die die Kommunikation steuert, überwacht und zugehörige Berechnungen anstößt.

### 5.3.1 Central Scheduler / Kernel

Um die zentrale Steuerung des Gesamtsystems zu übernehmen wird an dieser Stelle ein weiteres Modul konzipiert: Der „Kernel“ oder „Central Scheduler“. Er fungiert als zentrales Element im Visualisierungssystem. Von ihm werden alle Aktionen gesteuert und koordiniert, d.h. er wickelt den Hauptteil der Inter-System Kommunikation ab. Sämtliche von außen eingesteuerten Änderungen (z.B. das Verschieben einer Probe, die Änderung eines Visualisierungsparameters, das Laden eines neuen Datensatzes, etc.) laufen bei ihm in einer Nachrichtenliste (Event Queue) auf. Mit den Rest des Systems kommuniziert er über parametertransportierende Events - so genannte „Actions“ (siehe Kapitel 5.3.1.1). Diese Art der Anbindung des Kernels an den Rest des Systems erlaubt es, ihn asynchron von den restlichen Komponenten zu betreiben. Auf diese Weise kann man ihn von Dingen wie der Graphischen Ausgabe und z.B. der GUI abkoppeln.

Der wesentliche Arbeitsschritt des Central Schedulers besteht nun darin, aufgelaufene Events abzuarbeiten, gegebenenfalls neue zu generieren und an andere Systemkomponenten zu verschicken. Diese Events sind durch zwei verschiedene Vorgänge auslösbar:

**Automatisch generierte Events**
: Automatisch generierte Events sind Vorgänge, die vom System selbst ausgelöst werden. Ein einfaches Beispiel hierzu ist das folgende: Ein Timer löst in regelmäßigen Abständen das Weiterschalten der gesamten Szene von einem auf den nächsten Zeitschritt aus. Dadurch werden alle aktiven Visualisierungsmethoden dazu veranlaßt, die Darstellung (z.B. Strömungslinien) aus dem darauffolgenden Simulationzeitschritt zu errechnen. Wird dieser Vorgang für jeden vorhanden Zeitschritt wiederholt, entsteht bei ausreichender Gesamt-Systemgeschwindigkeit der Eindruck einer Animation.

**Manuel generierte Events**
: Zu den manuel generierten Events zählen in erster Linie vom Anwender ausgelöste Steuerbefehle. Ein einfaches Beispiel hierzu ist das gezielte manuelle Weiterschalten der Darstellung auf einen konkreten Zeitschritt, oder die definierte Änderung eines Parameters einer Probe (z.B. Länge oder Farbe der Strömungslinie).

Alle Events, d.h. automatisch und manuel generierte, laufen in derselben Message Queue des Central Schedulers zusammen. In Abbildung 5.7 ist skizziert, wie das im Einzelfall abläuft: Von „außen“, d.h. außerhalb des Kernels, wird ein Änderungswunsch per Event generiert. Das geschieht z.B, wenn der Anwender auf ein Element in der Benutzeroberfläche klickt, um

eine neue Probe/Visualisierungsmethode zu aktivieren oder einen Paramter einer vorhandenen zu verändern, oder ein Timer löst automatisiert das Weiterschalten des dargestellten Zeitschrittes aus. Der Central Scheduler prüft, ob das Event eine Neuberechnung eines Teiles der Daten notwendig macht. Falls ja, löst er die entsprechenden Berechnungen aus. Nach Abschluss der Berechnungen veranlaßt der Scheduler die Neugenerierung der entsprechenden graphischen Darstellung der Visualisierungsmethode (ein neuer Szenegraphknoten pro Methode wird erzeugt). Am Ende des Vorganges wird der alte Knoten mit dem so entstandenen neuen Knoten ausgetauscht. Sind mehrere Visualisierungsmethoden aktiv, wird durch die sortierte Abarbeitung der Nachrichten-Warteschlange gewährleistet, dass die Events in der sortierten Reihenfolge, in der sie auflaufen, auch abgearbeitet werden. So wird sichergestellt, dass erst auf die Darstellung eines neuen Zeitschrittes umgeschaltet wird, wenn die Darstellungsevents des vorherigen Zeitschrittes komplett von allen aktiven Visualisierungsmethoden erfolgreich abgearbeitet wurden.

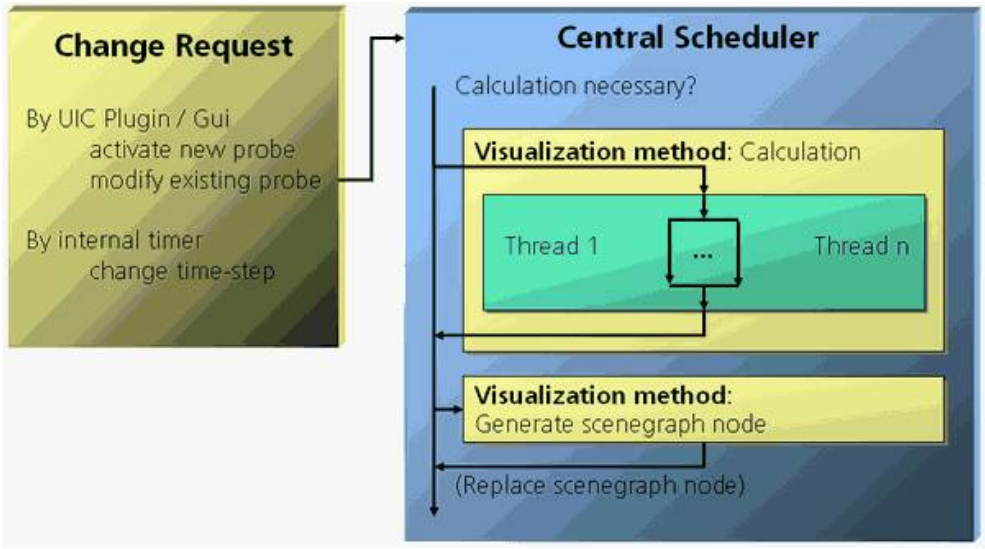



***Abbildung 5.7:*** *Funktionsweise des Central Schedulers: Events erreichen den Scheduler, der im Bedarfsfall die einzelnen Visualisierungsmethoden zu einer Neuberechnung ihrer Daten veranlaßt und anschließend die entsprechende graphische Darstellung aktuallisiert. Die blauen Teile werden durch Funktionen abgedeckt, die direkt im Central Scheduler verankert sind, die gelben Teile sind in der Visualisierungsmethode der betroffenen Probe zu finden.*

Um diesen Vorgang noch einmal genauer zu betrachten, hier nochmal ein entsprechendes Flußdiagramm (Abbildung 5.8). Sobald ein Event von außen hereinkommt wird vom Central Scheduler geprüft, ob eine Neuberechnung der Daten nötig wird. Falls nein, wird der Vorgang sofort wieder beendet. Falls ja, wird geprüft, ob für die betroffene Probe bereits eine Berechnung läuft. Falls ja, werden die auflaufenden Änderungswünsche so lange zwischengespeichert und zusammengefaßt, bis die laufende Berechnung beendet ist. Falls nein, wird eine neue Berechnung ausgelöst: Mehrere Threads führen eine entsprechende Neuberechnung der Visualisierungsmethode durch. Nachdem die Berechnung beendet ist, wird ein neuer Szenegraphknoten generiert und der alte mit diesem neuen ersetzt.

Damit dieser Ansatz reibungslos funktionieren kann, ist es wichtig, dass die Nachrichtenliste groß genug dimensioniert wird, um viele schnell aufeinanderfolgende Events aufnehmen zu

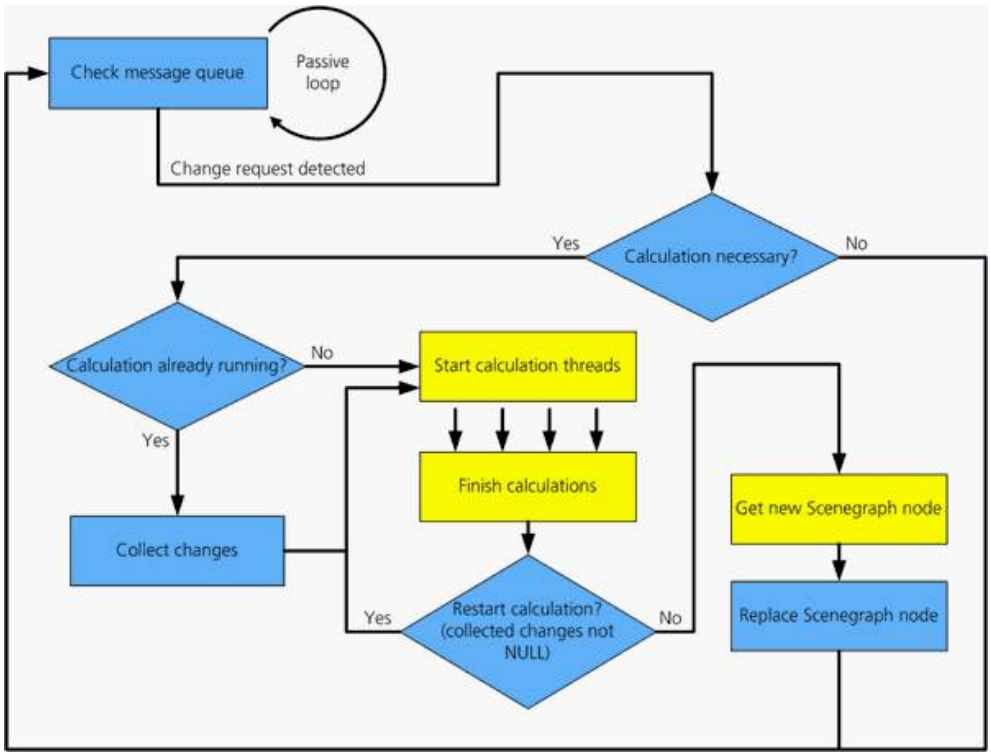



***Abbildung 5.8:*** *Flußdiagramm zum Central Scheduler.*

können bzw. der Central Scheduler auch schnell genug ist, die Nachrichtenliste auszulesen und die Events weiterzuleiten. Da der Central Scheduler im gesamten System als Modul zur Laufzeit nur einmal vorhanden sein darf, wird er als Singleton 6.2.1 entworfen.

#### 5.3.1.1 Abgekoppelte „Action"-basierte Kernel-Kommunikation

Wie im Kapitel vorher bereits erwähnt, kommunizieren die einzelnen Module des Systems über einen (asynchronen) Nachrichtenaustausch. Falls nötig, werden entsprechende Events generiert und von einem Modul zum anderen verschickt, das diese in einer Message Queue zur Auswertung sammelt. Die Nachrichten bzw. Events, die hierbei generiert werden, transportieren neben dem eigentlichen Namen der Nachricht zusätzliche Parameter. Diese speziellen Events werden in dieser Arbeit als „Actions" bezeichnet.

**Definition 5.5 (Action)** *Als Action werden Events bezeichnet, die im Visualisierungssystem zwischen einzelnen Modulen verschickt werden. Diese Events transportieren zum einen den Funktionsaufruf im entsprechenden Empfängermodul, der sich dahinter verbirgt, und zum anderen die dafür benötigten Aufrufparameter. Wird das Event aus einer Message Queue eingelesen, lässt sich das damit verbundene Kommando auswerten und die zugehörigen Parameter abfragen und weiterverwerten.*

Um die Speicherung der Actions in einer Nachrichtenliste / Message Queue zu ermöglichen, werden diese alle von einer verallgemeinerten Mutterklasse abgeleitet (siehe Abbildung 5.9)

- ähnlich wie das schon bei den Visualisierungsmethoden der Fall ist. Bei jeder im System verwendete Action werden so nach der Vererbung die zentralen Funktionen innerhalb der eigenen Klasse überladen. Um das auszuführende Kommando mit in die versendete Action zu kapseln, kommt das Command Design Pattern [Ale01, Mey98] zum Einsatz.

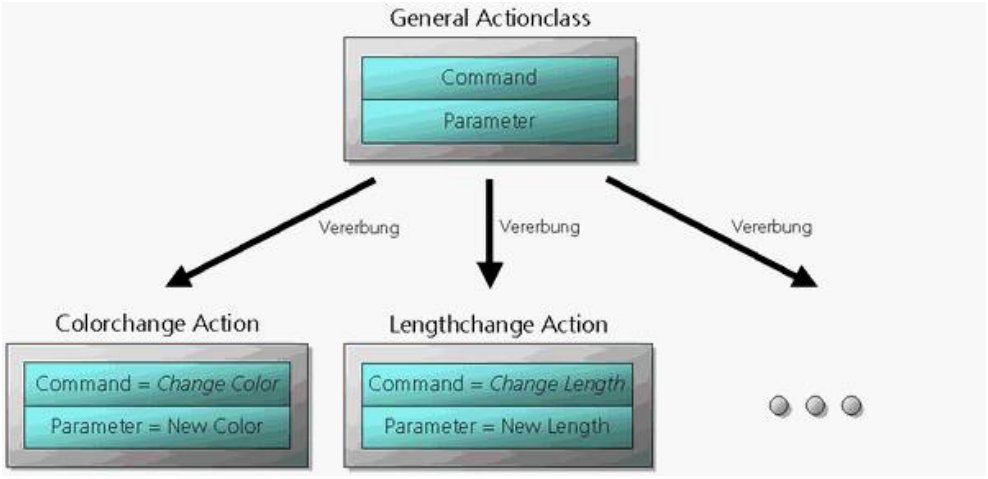



***Abbildung 5.9:*** *Actions: Abgeleitet von einer allgemeinen Mutterklasse transportieren die „Action" genannten Events sowohl das auszuführende Kommando als auch die entsprechenden Parameterwerte.*

Auf diese Weise wird es z.B. möglich, Parameteränderungen von der graphischen Benutzeroberfläche zur eigentlichen Visualisierungsmethode zu verschicken: Der Benutzer klickt auf ein Steuerelement im entsprechenden Teil der GUI, das die Länge von Strömungslinien festlegt. Anschließend wird eine entsprechende „Längenänderungsaction" vom GUI erzeugt und mit den Parametereinstellungen für die neue Länge versehen. Diese Action wird dann vom GUI (mit Hilfe des Kernels) an die betroffene Strömungslinien-Visualisierung verschickt (siehe Abbildung 5.10).


***Abbildung 5.10:*** *Beispiel-Action: Von der graphischen Oberfläche wird eine Längenänderung der Strömungslinien per Action an die entsprechende Visualisierungsmethode gesendet.*

#### 5.3.1.2 Ereignisgesteuerte Berechnung

Wie man im vorherigen Unterkapitel erkennen kann, wird es mit dem vorgestellten Konzept möglich, Berechnungen innerhalb des Systems immer nur dann anzustoßen, wenn sie nötig sind. Erst, wenn z.B. eine Parameteränderung ausgelöst wurde, wird eine entsprechende Action generiert, die dann nach Interpretation durch den Central Scheduler eine Berechnung auslöst. In dieser Arbeit wird dieses Vorgehen als „ereignisgesteuerte Berechnung" bezeichnet.

**Definition 5.6 (Ereignisgesteuerte Berechnung)** *Als ereignisgesteuerte Berechnung wird das Vorgehen bezeichnet, wenn Rechenschritte nur dann gestartet werden, wenn sie durch ein bestimmtes Ereignis ausgelöst werden.*

Die Algorithmen der Visualisierungsmethoden starten also nicht automatisiert von alleine zu bestimmten Zeitpunkten, sondern nur dann, wenn ein Ereignis stattgefunden hat, das die jeweilige Berechnung auslöst. Auf diese Weise kann man die Systemlast optimieren, da nur dann der Prozessor belastet wird, wenn auch ein entsprechendes Ereignis, d.h. ein „Grund" für die Berechnung, vorliegt.

Ereignisse, die eine neue Berechnung der Visualisierung erzwingen, sind:

**Datenänderungen**
: Die zugrundeliegenden Daten ändern sich. Dies geschieht z.B., wenn ein Datensatz gelöscht oder ein neuer geladen wird. Es ist auch dann der Fall, wenn neue Strömungsdaten über eine Netzwerkanbindung eintreffen.

**Parameteränderungen**
: Ein oder mehrere Parameter der Visualisierungsmethode wurden verändert (z.B. die Länge der Strömungslinien). Die entsprechende Berechnung muss mit der geänderten Linienlänge erneut durchgeführt werden.

**Probenänderungen**
: Die Probe, an die die Visualiserungsmethode gekoppelt ist, wurde verändert. Das geschieht, wenn die Probe ihre Position, Orientierung oder Größe ändert. Die an sie gekoppelte Visualisierung findet daraufhin in einem geänderten Ausschnitt aus dem Datenraum statt und muss neu gestartet werden.

#### 5.3.1.3 Abkopplung vom Szenegraph/Graphik-Format

Wie am Anfang dieses Unterkapitels beschrieben, generieren die Visualisierungsmethoden ihre geometrische Darstellung nicht direkt im Ausgabeformat (vgl. Abbildung 5.8). Zuerst werden die mathematischen Algorithmen auf das Datenfeld angewendet, um die nötigen Vorberechnungen für daran anschließende Visualisierung durchzuführen. Erst danach wird - auf Anfrage durch den Central Scheduler - von der jeweiligen Visualisierungsmethode der entsprechende Szenegraph Knoten aus diesen Ergebnisdaten generiert.

An dieser Stelle wird die eigentlichen Berechnung der Ausgabegeometrie vom Rest der Visualisierungsmethode abgekoppelt und ausgelagert. Über eine abstrakte Schnittstelle und einen verallgemeinerten Szenegraph-Knotentyp wird dann der entsprechende Darstellungsknoten erzeugt. Das in Kapitel 5.2.2 vorgestellte generische Klassenkonzept für Visualisierungsmethoden wird entsprechend erweitert (siehe Abbildung 5.11): Alle Berechnungsteile bleiben in der Visualisierungsmethode gespeichert, nur die Erzeugung des zugehörigen Szenegraphknotens wird ausgelagert.

So abgekapselt, wird es möglich den Teil der Visualisierungsmethode, der die graphische Darstellung generiert, schnell gegen einen anderen auszutauschen, den Berechnungsteil jedoch

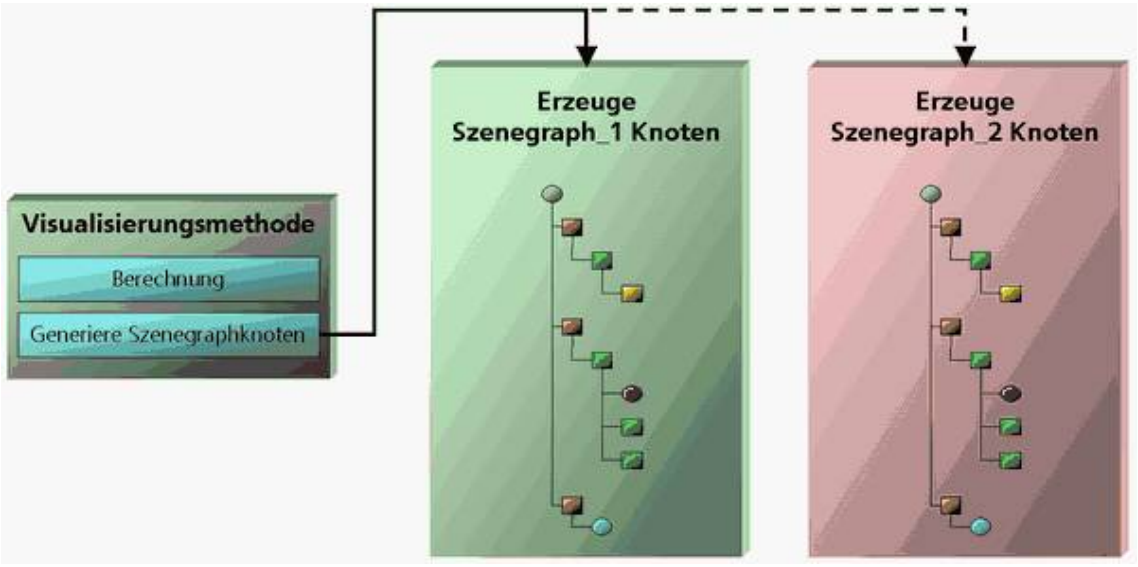



***Abbildung 5.11:*** *Die Erstellung der graphischen Repräsenation wird vom Rest der Visualisierungsmethode abgekapselt. Die beiden Farben des „Erzeuge Szenegraphknoten"-Teils repräsentieren zwei verschiedene Szenegraph-Formate, d.h. zwei unterschiedliche Ausprägungen dieser Funktionalität, die je nach Bedarfsfall austauschbar sind.*

beizubehalten. Damit wird der Wechsel von einem Graphik/Szenegraph Format zu einem anderen vereinfacht. In diesem Fall muss nur der ausgelagerte Szenengraph-Teil entsprechend angepaßt werden.

### 5.3.2 Parametersteuerung

Mit den in Kapitel 5.3.1.1 vorgestellten Actions lassen sich die Parameter, die das System bzw. die Visualisierungsmethoden beeinflussen, als Nachrichten verschicken. Die Parametersteuerung ist eine wichtige Anforderung im Visualisierungssytem (vgl. Kapitel 4.1.5.3). Wie bereits erwähnt, zeichnen sich unterschiedliche Visualisierungsmethoden dadurch aus, dass sie viele unterschiedliche Parameter zur Steuerung zur Verfügung stellen: Sind es z.B. bei einer Strömungslinienvisualisierung die Anzahl und die Länge der Linien, so sind es bei einer Iso-Flächen Darstellung die einzelnen Iso Werte oder beispielsweise ob ein Marching Cubes oder ein Tetraeder basiertes Verfahren zur Extraktion der Flächen verwendet werden soll.

Um den Umgang mit dieser abstrakten Schnittstelle der Visualisierungsmethoden zu vereinfachen, bieten sich zwei Konzepte an, die im Rahmen dieser Arbeit realisiert wurden:

**Parsersteuerung**

Zum einen kann man die Visualisierungsklasse so erweitern, dass sie selbst Auskunft darüber geben kann, welche Parameter in welcher Form sie vom System erwartet. Hier wird ein abstraktes Interface benötigt, dass dann beispielsweise durch einen Parser ausgewerten werden kann. Das System fragt die möglichen Parameter ab, stellt fest,

welcher davon wie geändert werden soll und schickt die gewünschte Änderung per Action an den Kernel / die zugehörige Visualisierungsmethode.

**Panelsteuerung**
Die andere Möglichkeit besteht darin, jede Visualisierungsmethode mit einem speziell für sie ausgelegten Fenster zu versehen, dass dann in die graphische Benutzeroberfläche im Bedarfsfall eingeblendet werden kann. Hier wird, ähnlich wie schon beim Szenegraph, der eigentliche Teil, der von konkreten Befehlen einer speziellen Graphikbibliothek abhängig ist, in ein extra Modul ausgelagert. Dort stellt eine Factory 6.2.2 auf Wunsch für das System das entsprechende GUI Fenster oder auch *Panel* bereit. Dadurch wird auch hier die Möglichkeit gewährleistet, im Bedarfsfall jederzeit schnell auf eine andere GUI Bibliothek ausweichen zu können, indem man einfach die Panel Factory entsprechend anpasst. Klickt der Anwender schließlich auf ein Eingabeelement im Panel, so wird wieder die entsprechende Action generiert und an den Kernel / die zugehörige Visualisierungsmethode verschickt.

## 5.4 Datenverwaltung

Eine wichtige Komponente des Gesamtsystems ist die Datenverwaltung. Ihre Aufgabe ist es, dem Rest des Systems die Strömungs-Simulationsdaten zur Verfügung zu stellen, die vom Anwender geladen wurden. Dabei muss sie verschiedene Randbedingungen beachten, wie Speichergröße, Parallelisierung und den damit verbundenen Thread-sicheren Zugriff. Natürlich muss der Datenzugriff auch schnell und einfach möglich sein, d.h. Effizienz spielt eine entscheidende Rolle bei der Konzeption dieses Moduls. Im Anschluß sind die zentralen Bestandteile des Datenverwaltungskonzeptes beschrieben.

### 5.4.1 Datenquellen

Daten, die zur Laufzeit der Visualisierungsumgebung im Speicher gehalten werden, werden in einer speziellen Struktur abgelegt, den „Datenquellen“ (s.u.). Mit Hilfe dieser Datenquellen haben die Visualisierungsmethoden/Proben die Möglichkeit, über einen einheitlichen Weg auf die Simulationsdaten zuzugreifen. Um die Simulationsdaten zur Laufzeit im Arbeitsspeicher des Visualisierungssystems zu halten, wird an dieser Stelle ein spezielles Konzept mit dem Namen „Datenquellen“ eingeführt.

**Definition 5.7 (Datenquellen)** *Datenquellen sind Container für die darzustellenden Simulationsdaten und abstrahieren die Struktur der zugrunde liegenden Daten. Sie enthalten eine standardisierte Schnittstelle, die zum einen Auskunft über den Typ, die Lage im Simulationsdatenraum und die Auflösung bzw. Größe der Daten zurückliefern kann. Ferner beinhalten die Datenquellen eine eigene Interpolationsroutine auf dem gekapselten Gitter.*

In einem „Container“ genannten Bereich wird - pro geladenem Datensatz - eine Datenquelle abgelegt. Möchte der Nutzer nun verschiedene Visualisierungemethoden auf diesen Datensatz anwenden, so hat das System die Möglichkeit, zunächst deren Lage und Ausdehnung

im Simulations(Zeit)Raum und deren Typ abzufragen. Anschließend wird dann eine Verbindung zwischen einer Datenquelle und einer Visualisierungsmethode hergestellt. Die entsprechenden Zahlenwerte für die Berechnung werden über die standardisierte Schnittstelle aus der Datenquelle gelesen. Die Schnittstelle wird hierbei so ausgelegt, dass die Visualisierungsmethode lediglich unter Angabe einer Raum/Zeit Koordinate $(x, y, z, t)$ die Datenwerte abfragen kann. Hierbei geschieht die notwendige Interpolation automatisch (transparent) im Hintergrund, falls der abgefragte Wert nicht genau auf einem Gitterpunkt der Datenquelle liegt: Die zugehörige Zelle im Datengitter wird identifiziert und der Wert aus den Randpunkten/Kanten/Zellen ermittelt. Auf diese Weise können die Visualisierungsmethoden mit beliebig strukturierten Datenquellen zusammenarbeiten, ohne deren spezielle Gegebenheiten (z.B. spezielle Gitterstrukturren) beachten zu müssen. Dadurch können leicht neue Datenquellen mit einer speziellen Struktur der Daten in das System ergänzt werden, ohne dass die vorhandenen Visualisierungsmethoden angepasst werden müssen. Die notwendige Interpolation von umliegenden Gitterpunkten auf die angegebene Koordinate geschieht hierbei transparent im Hintergrund (Abbildung 5.12)

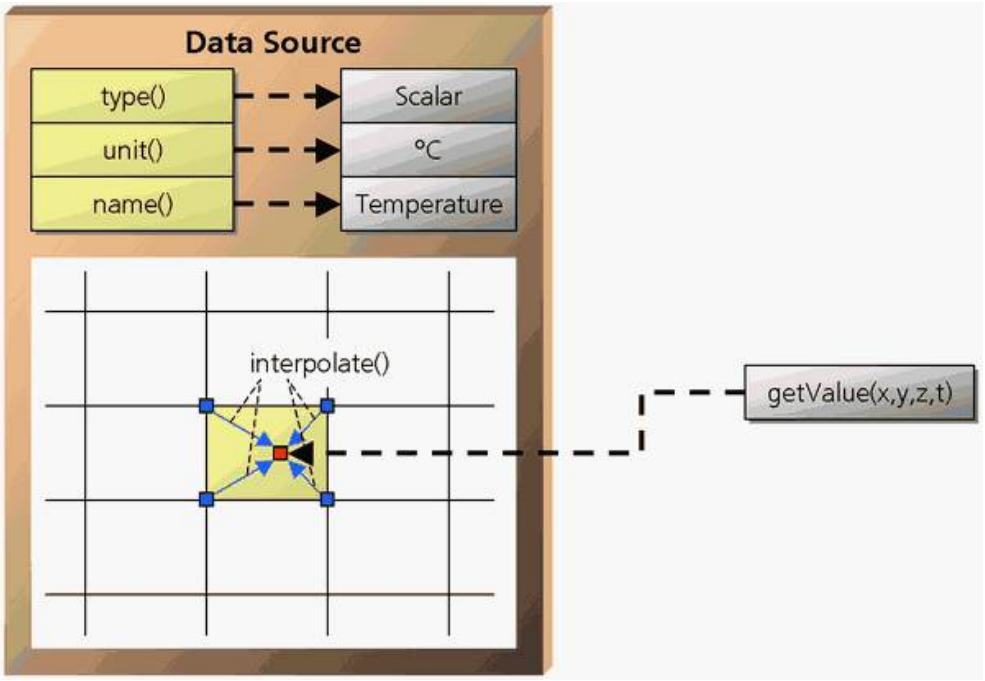



***Abbildung 5.12:*** *Neben Informationen z.B. bezüglich ihres Types, Namens und der Maßeinheit bietet die Datenquelle auch eine transparente Interpolation: Von außen wird lediglich die $(x, y, z, t)$ Koordinate angegeben, die Datenquelle ermittelt den relevanten Wert der physikalischen Größe an der vorgegebenen Stelle.*

Wenngleich die wichtigsten Datentypen für Strömungssimulationen in dreidimensionalen Skalar- und Vektorfeldern vorliegen, wird das System dennoch so angelegt, dass sich weitere Datentypen im Bedarfsfall einfach durch Vererbung & Überladen der zentralen Funktionen ergänzen lassen.

Das Bereitstellen der Typinformationen der Daten in den Datenquellen ist notwendig, da Visualisierungsmethoden vom System nur denjenigen Datenquellen zugeordnet werden dürfen,

die sie auch verarbeiten können. Eine Visualisierung von Strömungslinien kann beispielsweise nicht sinnvoll auf einem Skalarfeld durchgeführt werden. Die Typunterscheidung ist hier sehr wichtig. Zusätzlich lassen sich noch Meta Daten, wie „Name der repräsentierten physikalischen Größe“ (z.B. „Temperatur“) oder „Maßseinheit“ (z.B. „Meter“ oder „C°“) mit über diese Schnittstelle bereitstellen. Mit dieser Grundlage hat man später im System die Möglichkeit, die Visualisierungsmethoden über ein Prozeduales Interface mit den Datenquellen zu verbinden, wie es bereits ähnlich in [RSS96, COV] umgesetzt wurde.

### 5.4.2 Zentrale Datenverwaltung

Um diese Datenquellen zu verwalten, benötigt man eine zentrale Instanz, den sog. „Datamanager“. Er kennt alle verfügbaren Datenquellen und ermöglicht den zentralen Zugriff auf sie. Er koordiniert das Löschen (*remove*) und Erstellen (*add*) von Datenquellen und überwacht den Datenfluß (siehe Abbildung 5.13).

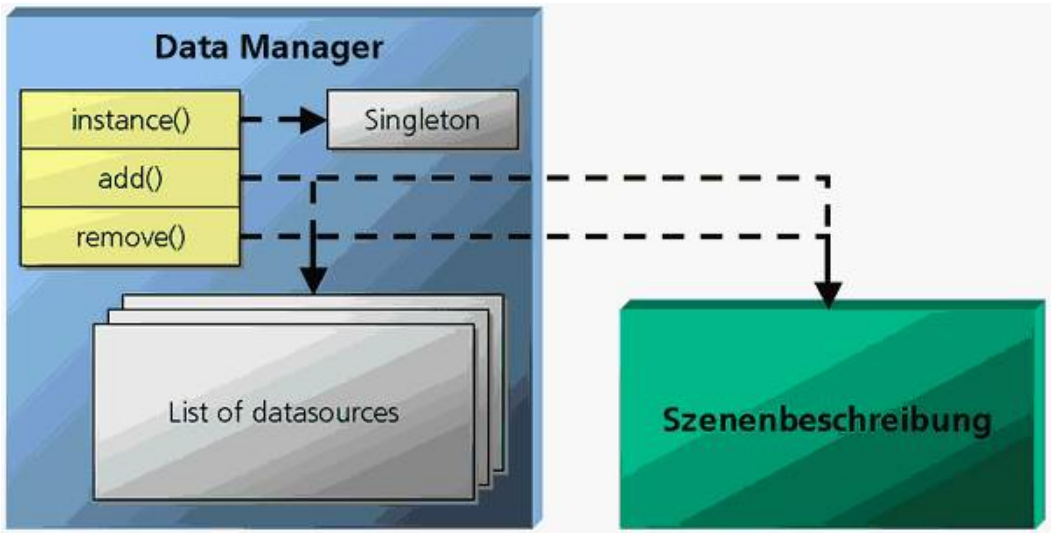



***Abbildung 5.13:*** *Datamanager: Als zentrale Instanz verwaltet er die zur Verfügung stehenden Datenquellen für das System. Neue Datentypen werden bei seiner Factory registriert.*

Genau wie der Central Scheduler ist er nur einmal im System vorhanden und wird als Singleton (*instance*) angelegt. Für die Erzeugung von neuen Datenquellen ist eine im Datamanager enthaltene Factory verantwortlich. Über sie kann das System leicht um neue Datenquellentypen (neben Skalar und Vektor) ergänzt werden (z.B. Tensor). Bei dieser Factory werden die Erzeuger für alle verfügbaren Datenquellentypen registriert und können anschließend vom System verwendet werden, um neue Instanzen von Datenquellen entsprechenden Typs zu erzeugen.

### 5.4.3 Sicheres Multithreading

Die Datenquellen, die im Visualisierungssystem verwendet werden, können zur Lauftzeit angelegt und/oder gelöscht werden. Ferner wird es durch eine spezielle Speicherstruktur möglich sein, Teilbereiche der Daten nachzuladen und andere dafür auszulagern (siehe Kapitel 5.6). Sie sind sozusagen „dynamisch“, da sich der Datenbestand im Betrieb verändern kann. An dieser Stelle wird es nötig, die Kommunikation mit den Daten für parallele Zugriffe abzusichern. Damit die dynamischen Datenquellen sicher mit mehreren Threads verwendet werden können, sind Synchronisationsmechanismen für den Zugriff auf die Daten erforderlich. Der Zugriff auf die Daten beinhaltet das aus [Tan92] bekannte Leser / Schreiber Problem.

Bei der Lösung dieses Problems ist es wichtig, mehreren parallelen Threads (d.h. beispielsweise verschiedenen zur Laufzeit aktiven Visualisierungsmethoden) das gleichzeitige Lesen der Daten zu erlauben. Die Visualisierungsmethoden verwenden durchweg alle nur lesenden Zugriff auf die Daten: Sie fragen die physikalischen Werte der untersuchten Datenmenge an vorgebenen Raumpunkten ab und starten daraufhin die Berechnung ihrer Visualisierung. Die einzige Instanz, die schreibenden Zugriff auf die Daten hat, ist der Datamanager selbst. Über ihn hat das System die Möglichkeit, Daten zu löschen oder aus/einzulagern. Auch hier muss ein entsprechender Synchronisationsmechanismus integriert werden. Der Schreib- bzw. Änderungsvorgang an den Daten ist exklusiv und kann nicht, wie die Lesevorgänge, parallel von mehreren gleichzeitig ausgeführt werden. Darüber hinaus muss das Lesen der Daten verhindert werden, während sie vom System bzw. Datamanager verändert werden. Da er allerdings die einzige Systeminstanz ist, die die Daten verändert, ist die Synchronisation an dieser Stelle sehr leicht möglich. Dennoch wird der Datamanager so angelegt, dass auch mehrere Schreiber unterstützt werden.

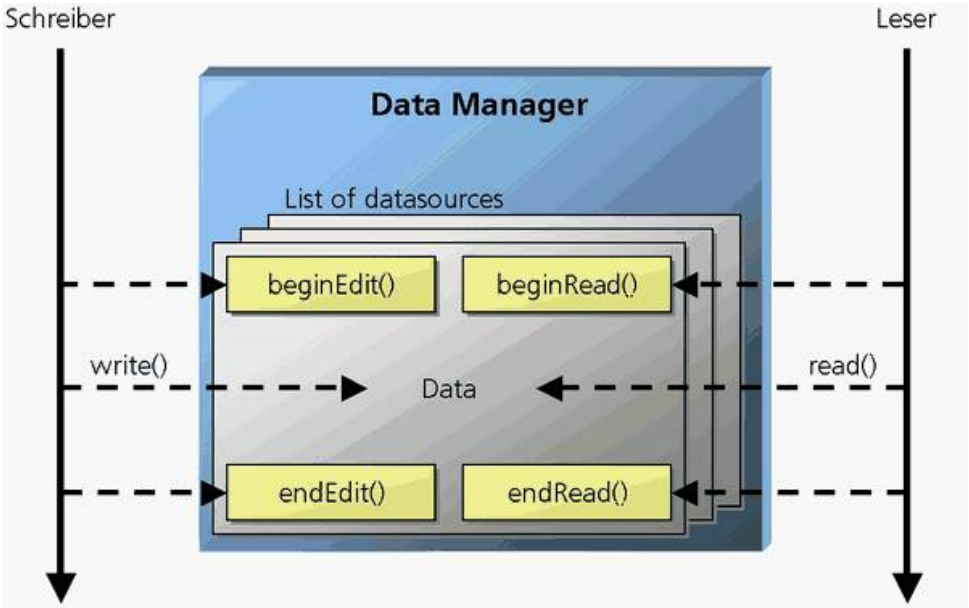



***Abbildung 5.14:*** *Leser / Schreiber Synchronisation: Über entsprechende Funktionsaufrufe sichern sich Leser und Schreiber im System dem Zugriff auf die Daten bzw. melden sich beim Datamanager und der entsprechenden Datenquelle hierfür an.*

Um maximale Parallelität der vielen Leser im Zusammenspiel mit dem Data Manager und die Konsistenz der Daten zu gewährleisten, werden die folgenden vier Methoden angelegt (siehe Abbildung 5.14):

**beginEdit()**
: Ein Schreiber meldet das Verändern der Daten an. Er tritt in eine (passive) Warteschleife ein, bis er vom Datamanager hierfür die Berechtigung erhält. Der Schreibvorgang ist exklusiv, d.h. die Freigabe geschieht erst, wenn kein anderer Schreiber und keine Leser mehr auf die Daten zugreifen.

**endEdit()**
: Ein Schreiber meldet das Ende der Änderungsvorgänge am Datenbestand. Anschließend haben andere Schreiber die Möglichkeit sich diesen Zugriff zu sichern und die Leser können die Daten abfragen.

**beginRead()**
: Ein Leser meldet sich zum Abfragen der Daten an. Dieser Vorgang blockiert keine anderen Leser, d.h. es können sich jederzeit weitere Leser zusätzlich anmelden. Nur Schreiber müssen warten.

**endRead()**
: Ein Leser meldet das Ende der Abfrage der Daten. Er wird aus der Leserliste zu diesem Datenbestand entfernt.

Um für den Rest des Systems den Zugriff möglichst einfach zu halten, müssen lediglich Bereiche, die schreibend auf die Datenquelle zugreifen, mit beginEdit() und endEdit() geklammert werden. Bereiche, die lesend auf Daten zugreifen, werden mit beginRead() und endRead() geklammert. Auf diese Weise wird sichergestellt, dass kein Thread Daten verändert während ein anderer Thread Daten liest oder ebenfalls versucht, Daten zu schreiben. Da wie erläutert das parallele Auslesen der Daten unkritisch ist, wird den Lesern der mehrfache parallele Zugriff ermöglicht.

### 5.4.4 Transparentes nachladen und auslagern

Ein weiterer Ansatz, um die Leistungsfähigkeit auf schwächeren System zu steigern, ist es, Datenteile, die momentan nicht benötigt werden, während der Laufzeit auszulagern. So wird Speicher gespart und man kann auch Untersuchungen an Datensätzen durchführen, die zu groß wären, um sie komplett in den Speicher zu laden. Jedoch muss in diesem Fall der untersuchte Teilbereich und/oder die geladene Auflösung klein genug gehalten werden. Führt man jedoch eine solche Auslagerungsfähigkeit in das System ein, wird es wiederum auf der anderen Seite nötig, dass man auch Teile im Bedarfsfall direkt nachladen kann.

Ein solches Vorgehen ist beispielsweise bei großen Geometrieszenen eine gängige Technik [Sah03], um komplexe Szenen auch auf leistungsschwachen Endgeräten bewältigen zu können, die für eine Komplettdarstellung zu umfangreich wären. Eine Übertragung dieses Prinzips auf Strömungs- / Volumendatensätze garantiert äquivalente Erfolge.

Die Nachlade- und Auslagerungsvorgänge werden im Datamanager als (interne) schreibende Zugriffe gehandhabt: Ein Edit Vorgang wird angemeldet. Nachdem die Leser (= Visualisierungsmethoden) ihre Arbeit beendet haben, hat der Datamanager die Möglichkeit, entsprechende Teile des Datenbestandes zu entfernen oder neu hinzugekommene einzufügen. Wichtig hierbei ist es, dass die Datenquelle nach dem Ende der Editvorgänge jederzeit „vernünftige“ Werte für den gesamten Datenvolumen-Raum liefern kann, wenngleich sie auch ungenau bzw. nicht detalliert in „uninteressanten“ (= ausgelagerten) Gebieten sein können.

Auf die genaue Vorgehensweise bzgl. der Ein- und Auslagerung der Volumendaten wird im Kapitel über Progressive Gitter (Kapitel 5.6) weiter eingegangen.

## 5.5 Skalierbarkeit

Wie in Kapitel 4.2.10 beschrieben, ist die weitestgehende Leistungsunabhängigkeit von der zugrundeliegenden Hardware eine wichtige Eigenschaft eines Visualisierungssystems. Um dieses Ziel zu erreichen, wird das Konzept eines Visualisierungssystem, das in dieser Arbeit entwickelt wird, skalierbar ausgelegt.

Skalierbarkeit heißt in diesem Zusammenhang, dass die einzelnen Komponenten / Module in der Lage sind, sich an unterschiedliche Begebenheiten anzupassen. Im Falle der Datenhaltung heißt das konkret, dass die Bereiche, in denen die volle / höchstmögliche Datenauflösung im Speicher gehalten werden, entsprechend kleiner ausfallen. Das kann soweit gehen, dass selbst in den direkt untersuchten Bereichen nicht die volle zur Verfügung stehende Datenauflösung in den Speicher geladen wird, da diese hierfür unter Umständen bereits zu groß ist. Durch die progressive Datenstruktur (siehe Kapitel 5.6) wird es möglich, auch bei wenig Speicher noch adäquate Ergebnisse zu visualisieren.

Neben der reinen Speichermanagement-Funktionalität ist die adaptierbare Parallelisierung der wichtigste Aspekt für die Skalierbarkeit des Visualisierungssystems. Mit Hilfe von parallelisierten Komponenten ist man in der Lage, auf Mehrprozessorsystemen weitere Leistungssteigerungen zu erreichen. Die Anzahl der hierbei zu verwendenten Threads wird dabei variabel gewählt, d.h. während Laufzeit wird erst festgelegt, wieviel Threads z.B. für die jeweilige Visualisierungsmethode genau verwendet werden. Im nächsten Unterkapitel wird dieser Aspekt genauer ausgeführt.

### 5.5.1 Parallelisierung

Zur Umsetzung der Skalierbarkeit wird das Visualisierungssystem parallelisiert. Betrachtet man die verschiedenen Arten der Parallelisierung (siehe Kapitel 3.4.3), so muss man sich zwischen einer Distributed (getrennte Speicher und Prozesse) und einer Shared Memory (gemeinsamer Speicher und mehrere Threads) Architektur entscheiden. Diese Entscheidung hat mit den wichtigsten Einfluß auf das dahinterstehende Systemlayout und ist von grundlegender Bedeutung.

In einer Distributed Memory Architektur herrscht in der Regel ein großer Kommunikationsbedarf der einzelnen Komponenten. Da sie keinen gemeinsamen Adressraum haben, müssen sämtliche Daten auf Inter-Prozess Ebene transferiert werden (z.B. über ein Netzwerk oder die Festplatte). Diese Art der Kommunikation bringt vergleichsweise hohe Latenzen, d.h. Antwortzeiten, mit sich und stellt den wichtigsten Nachteil dieser Parallelisierungsart dar. Der große Vorteil hierbei ist, dass den einzelnen Prozessen und damit dem Gesamtsystem mehr Speicher zur Berechnung zur Verfügung steht (z.B. in einem Cluster) und somit größere Datenmengen bewältigt werden können. Durch die Parallelisierung auf Prozess Ebene ist auf Systemen, die aus mehreren Rechnern bestehen (Cluster), eine Leistungssteigerung des Gesamtsystems zu erwarten.

In einer Shared Memory Architektur steht in der Regel den einzelnen Threads weniger Speicher zur Verfügung. Die bewältigbaren Datenmengen sind entsprechend kleiner. Dafür entfällt die aufwendige und teure Kommunikation, da die Threads auf gemeinsame Puffer und Adressräume zugreifen können. Durch die Parallelisierung auf Thread Ebene ist auf Systemen, die mehrere CPUs zur Verfügung haben (Mehrprozessor Rechner), eine Leistungssteigerung des Gesamtsystems zu erwarten.

Um die richtige Wahl für das Visualisierungssystem zu treffen, muss man also zwischen der Möglichkeit, viel Speicher verbunden mit (teuerer) Inter-Prozess Kommunikation und der Möglichkeit, weniger Speicher aber (billige) Inter-Thread Kommunikation, wählen. Betrachtet man jetzt die Aufgabenstellung genauer, so wird folgendes klar: Während bei der Strömungssimulation im Vorfeld in der Regel viele unabhängige Teilaufgaben mit wenig Synchronisationsaufwand den Hauptschwerpunkt bilden, so ist bei der anschließenden Visualisierung genau das Gegenteil der Fall: Viele kleine Berechnungen (z.B. eine Menge an Strömungslinien) werden auf denselben Datensatz angewendet. Es herrscht reger Kommunikations- und Synchronisationsbedarf, Speicher wird hingegen nicht viel benötigt. Folglich ist mit einer Optimierung des Kommunikationspfades mehr Leistungssteigerung zu erwarten, als mit einer Vergrößerung des zur Verfügung stehenden Speichers. Aus diesem Grund ist die Shared Memory Architektur das in dieser Arbeit verwendete Parallisierungskonzept zusammen mit einer massiv parallelen Visualisierung auf Thread-Ebene. Mit ihr ist es auch möglich, auf normalen PCs das Visualisierungssystem einzusetzen ohne z.B. das Vorhandensein eines Clusters vorauszusetzen. Dennoch wird es mir Ihrer Hilfe erreicht, das System zu beschleunigen, sobald mehrere Prozessoren zur Verfügung stehen.

Wie bereits in Kapitel 3 erläutert, existiert mittlerweile die Möglichkeit, aus Clustersystemen einen „virtuellen Rechner mit vielen CPUs“ zu machen. So kann man Shared Memory parallelisierte Systeme auch auf Clustern betreiben. Zwar sind hier deutliche Verzögerungen durch die (zwar im Hintergrund - aber dennoch vorhandene) Netzwerkkommunikation zu erwarten, jedoch kann man so ohne große Schwierigkeiten den bereits threadweise parallelisierten Code auf den Cluster portieren.

Um eine sinnvolle Parallelisierung des gesamten Visualisierungssystems vorzunehmen gab es bereits vor dieser Arbeit zahlreiche Ansätze. Einer von ihnen beschäftigte sich beispielsweise mit der Idee, die Parallelisierung des Systems mit Hilfe einer Domänen-Dekomposition der Datenmenge im Objektraum vorzunehmen, d.h. auf funktionaler Ebene durchzuführen (z.B. Manna / Visa System [dP97]). Hierbei wird die Datenmenge bereits vor der Visualisie-

rung in verschiedene getrennte Bereiche aufgrund ihrer unterschiedlichen Objektzugehörigkeit aufgeteilt. Diese unterschiedlichen Bereiche werden dann parallelisiert bearbeitet. Dieses Prinzip läßt sich nicht direkt auf simulierte Strömungsdaten übertragen, da eine Domänen-Dekomposition anhand unterschiedlicher Bereiche im Datensatz mit zugehöriger Verwaltung für dieses Problem zum einen aufwendig und damit ineffizient wäre. Zum anderen zerfallen Strömungsdaten nicht in Domänen. Interpretiert man jedoch eine „Domäne“ beispielsweise als ein Feld einer physikalischen Größe in einem Datensatz, der mehrere solcher Felder enthalten kann, so wird ein äquivalentes Prinzip anwendbar, die Domänenidee läßt sich entsprechend umgestalten und übertragen: Für jedes dieser Felder (also z.B. Temperatur, Druck, Dichte, Windrichtung und -Stärke) wird eine eigene Datenquelle angelegt, die unabhängig von anderen Datenquellen bearbeitet / abgefragt werden kann. Der parallele Zugriff auf die Datenfelder wird möglich. Allerdings kann man weit über die reine Parallelisierung über die Domänen, d.h. Datenfelder in einem Visualisierungssystem hinausgehen, indem man auch den Rest des Systems konsequent parallelisiert.

Neben den beschriebenen Ideen besteht außerdem die Möglichkeit, eine Parallelisierung der Ausgabe durchzuführen, nachdem die dafür benötigte Ausgabegeometrie fertig berechnet wurde. Für diesen Ansatz existieren mittlerweile eine Reihe „schlüsselfertiger“ Systeme, die diese Aufgabe zufriedenstellend erfüllen (z.B. OpenSG [OPEc]). Wichtiger Entwicklungsgegenstand ist demnach die der Ausgabe vorgeschaltete Berechnung (Visualisierungsmethoden und Systemarchitektur). Hier besteht noch großes Verbesserungspotential durch Integration effektiver Parallelisierungskonzepte.

Die Idee dieser Arbeit ist es, die Parallelisierung auf drei verschiedenen Ebenen des Systems gleichzeitig durchzuführen, was im Folgenden kurz erläutert wird.

#### 5.5.1.1 Parallelisierung innerhalb einer Visualisierungsmethode

Ein wesentlicher Punkt einer Visualisierungsmethode ist es, dass sie aus einer Menge vorgegebener Daten mit Hilfe von mathematischen und algorithmischen Methoden Geometrien errechnet, die dann darstellbar sind. Neben dem reinen Speicherzugriff ist hierbei der Berechnungsvorgang die kritische Komponente. Selbst wenn der Umstand gegeben ist, dass Zugriffe auf den Arbeitsspeicher bzw. auf die Teile der Datenmenge, die relevant sind, hinreichend schnell erfolgen, so ist es immer noch wichtig, dass die Geometrie, die die Visualisierungsmethode aus diesen Daten errechnet, schnell genug generiert wird.

Um den Berechnungsvorgang einer Visualisierungsmethode zu beschleinigen, bietet es sich an ihn zu parallelisieren. Hier muss man zwischen zwei Arten der Parallelisierung unterscheiden:

**Parallelisierung des Algorithmus**

Hierbei ist zu beachten, dass bei vielen Visualisierungsmethoden grundsätzlich verschiedene Algorithmen zum tragen kommen, die jeweils auch auf unterschiedliche Weise parallelisierbar sind. Die Strömungslinien-Visualisierung beispielsweise errechnet den Weg mehrerer virtueller Partikel in einem Strömungsfeld. Hier bietet sich eine bestimmte Art und Weise der Parallelisierung an: Theoretisch könnte man für jeden dieser Partikel

eine eigene Berechnung starten, die von den Berechnungen anderer Partikel unabhängig, und damit in einem getrennten Thread, stattfinden kann. Der Einfachheit halber kann man hier z.B. Gruppen von Partikeln zusammenfassen und diese dann jeweils von einem Thread berechnen lassen (Abbildung 5.15).

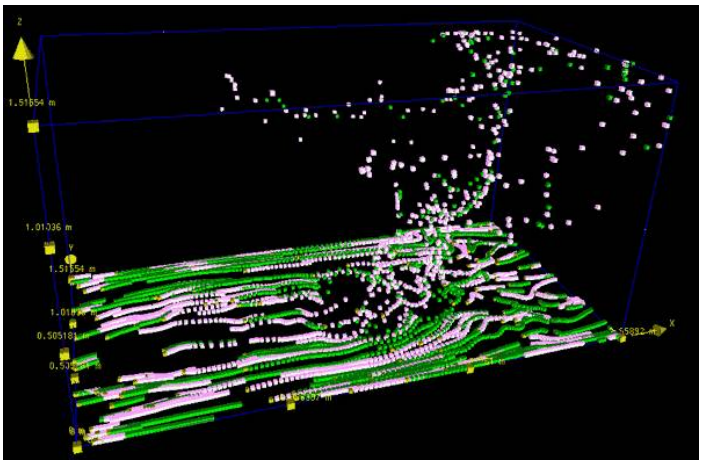


***Abbildung 5.15:*** *Partikel Visualisierungsmethode (siehe Kapitel 7.1.4) - Einfärbung nach verwendeten Threads (2)*

**Parallelisierung durch Datenraumaufteilung**

Bei dieser Parallelisierungsmethode wird statt des eigentlichen Algorithmus einfach der zur Untersuchung anstehende Datenraum räumlich aufgeteilt und die Ergebnisse der einzelnen Teile getrennt voneinander ermittelt. Eine Anpassung des eigentlichen Algorithmus ist nicht nötig. Diese Vorgehensweise ist nur anwendbar, wenn die Art des Algorithmus eine Zerlegung in räumlich getrennte, unabhängig voneinander berechenbare Teilbereiche zuläßt und nicht etwa die Ergebnisse des einen Datenraumteilbereiches die eines anderen direkt oder indirekt beeinflussen können. Betrachtet man eine Visualisierungsmethode, wie beispielsweise die Iso-Flächen Generierung mit Hilfe des Marching Cubes Algorithmus (vgl. Kapitel 7.1.6.1), so bietet sich diese Herangehensweise für die Parallelisierung an: Man teilt den Datenraum in unterschiedliche Segmente ein und übergibt jedem Berechnungsthread eines dieser Segmente (Abbildung 5.16).

Es bleibt also für jede Visualisierungsmethode eine eigene Aufgabe und Herausforderung, die mögliche Parallelisierungsart zu bestimmen und dadurch die zugehörigen Berechnungen auf mehreren Prozessoren optimal zu verteilen. Es gibt keinen pauschalen Ansatz, sondern es müssen individuelle Lösungen realisiert werden. Im Einzelfall ist zu entscheiden, wie man welchen Algorithmus einer Visualisierungemethode am effizientesten parallelisiert.

#### 5.5.1.2 Parallelisierung zwischen den Visualisierungsmethoden

Wie in Kapitel 4.1.2 beschrieben, ist es eine Anforderung an das Gesamtsystem, dass mehrere Visualisierungsmethoden gleichzeitig dargestellt werden können. Hierbei kann mit Hilfe von Parallelisierung eine weitere Beschleunigung erreicht werden.

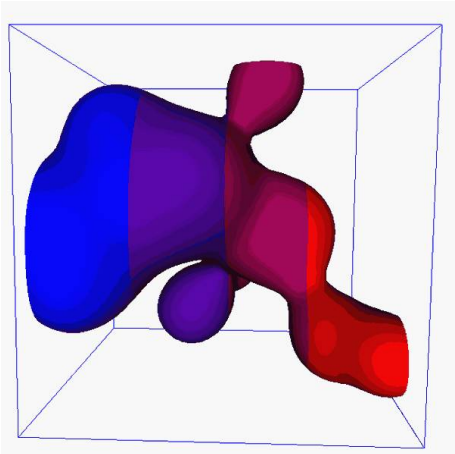


***Abbildung 5.16:*** *Marching Cubes Visualisierungsmethode (siehe Kapitel 7.1.6.1) - Einfärbung nach verwendeten Threads (4).*

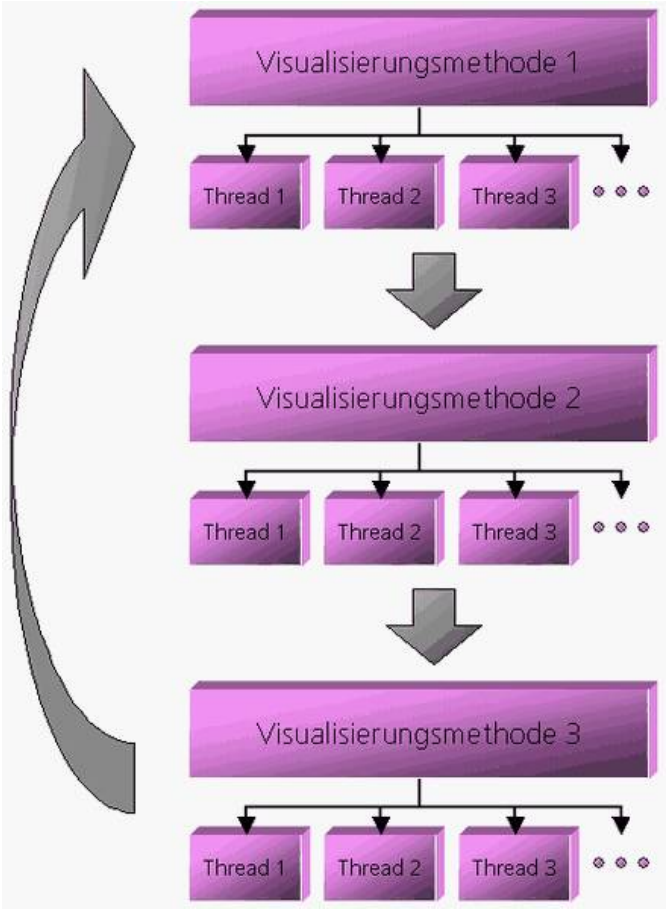



***Abbildung 5.17:*** *Parallelisierung zwischen Visualisierungsmethoden 1: Die Parallelisierung findet nur innerhalb der jeweiligen Visualisierungsmethode statt. Die einzelnen Methoden werden nacheinander in einer Schleife abgearbeitet.*

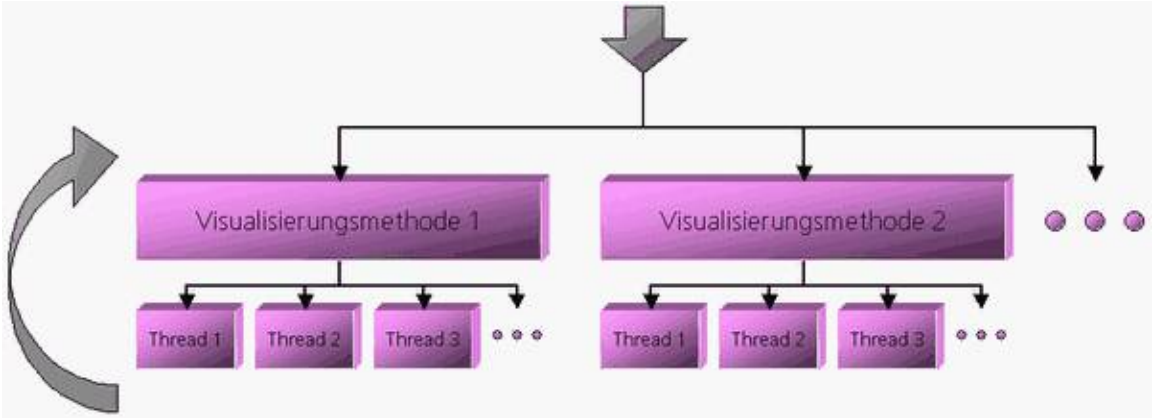



***Abbildung 5.18:*** *Parallelisierung zwischen Visualisierungsmethoden 2: Die Visualisierungsmethoden werden nicht nacheinander, sondern parallel abgearbeitet.*

Jede aktive Visualisierungsmethode bringt eine Berechnung der Ergebnisgeometrie mit sich. Ermöglicht man es nun, dass die aktivierten Visualisierungsmethoden alle gleichzeitig ihre Ergebnis Geometrien berechnen können, indem sie in jeweils eigenen Threads parallel arbeiten, so hat man eine weitere Stufe der Parallelisierung erreicht (Abbildung 5.17 und 5.18). Der in Kapitel 5.2.2.1 vorgestelle Visualization Method Pool wird in Anlehnung an diese Idee parallelisiert konzipiert.

#### 5.5.1.3 Parallelisierung des Gesamtsystems

Betrachtet man das Ergebnis der Visualisierungsmethoden, so entstehenden durch ihre Berechnungen Geometrien, die die entsprechenden Hervorhebungen bzw. Darstellungen im Datenfeld repräsentieren. Diese Geometrien werden vom System als Szenegraphknoten gespeichert. Neben der bereits vorgestellten Parallelisierung auf Knoten-Ebene (d.h. innerhalb einer Visualisierungemethode) und der Parallelisierung zwischen den Knoten (d.h. parallel bearbeitete Visualisierungsmethoden), dient jetzt als dritte Ebene der Parallelisierung das Gesamtsystem.

Das Visualisierungsssystem wird so angelegt, dass die einzelnen Module (d.h. Datenverwaltung, Central Scheduler, Visualization Method Pool, etc.) jeweils in getrennten Threads stattfinden. Das führt wieder zurück zur ursprünglichen Idee des modularen Aufbaus (vgl. Anfang dieses Kapitels) des Gesamtsystems. Zwischen den einzelnen Modulen, d.h. Threads, werden Nachrichten per Events ausgetauscht und sie laufen asynchron nebeneinander.

Dieser Teil der Parallelisierung beschreibt die dritte Ebene, nach der die Aufgaben in mehreren Threads auf die zur Verfügung stehenden Prozessoren verteilt werden (siehe Abbildung 5.19).

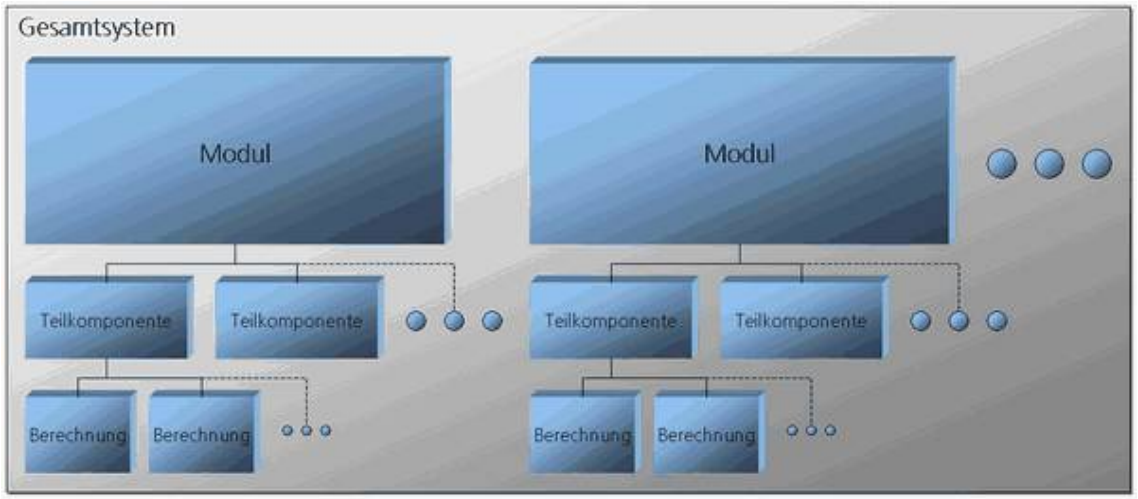



***Abbildung 5.19:*** *3 Ebenen der Parallelisierung: Neben der Parallelisierung auf Modulebene, wird das System auch noch innerhalb der Module und in den entsprechenden Teilkomponenten parallelisiert..*

## 5.6 Progressives Datenformat für CFD Daten

Die Optimierung des Datenzugriffs spielt bei einem Strömungs-Visualisierungssystem mit die wichtigste Rolle. Während es bei der Simulation in erster Linie darauf ankommt, möglichst schnell viele Daten einfach nur auf die Festplatte zu schreiben und die entsprechenden Speicher-/Dateiformate möglichst simpel gestrickt sind, kommt es bei der anschließenden Darstellung darauf an, möglichst effizient die relevanten Teile der Datenmasse zu identifizieren und zu laden. Hier spielt die intelligente Organisation der Daten eine entscheidende Rolle. Wie bereits in Kapitel 4.2.1 erläutert, wird diese Aufgabe um so schwieriger, je größer die Datenmengen werden und je kleiner der zur Verfügung stehende Arbeitsspeicher ist. An dieser Stelle wird ein intelligenter Kompressionsansatz (Kapitel 4.1.1) benötigt, der im Folgenden entwickelt wird. Die vielen unterschiedlichen Datenformate (vgl. Kapitel 3.5), die in der Strömungssimulation verwendet werden, sind hierfür nicht ausgelegt.

Hierbei muss das Problem gelöst werden, dass die Auflösung der Daten (z.B. in einem Gitter) nicht zu der Auflösung der Darstellung (z.B. der des Bildschirms oder eine Textur) passt. Folglich muss irgendwann zwischen der Datenerfassung bzw. -erzeugung und deren Visualisierung eine Konvertierung stattfinden. Generell lassen sich hierbei zwei verschiedene Herangehensweisen unterscheiden:

**Konvertierung vor der Visualisierungsberechnung**

Bei dieser Methode wird versucht, die original Gitter- bzw. Datenstrukturen, die die Simulation zur Berechnung der Strömungsdaten verwendet, möglichst lange auf dem Weg von ihrer Entstehung bis zu ihrer Darstellung / Auswertung beizubehalten. Das bedeutet im Einzelfall, dass man für die unterschiedlichsten Kombinationen von Gittertypen und Visualisierungsmethoden getrennte Algorithmen für die Visualisierung entwickeln muss (siehe auch 4.2.9).

Die wichtigsten Vorteile dieses Ansatzes lassen sich folgendermassen zusammenfassen:

- Die Genauigkeit der Eingangsdaten wird so lange wie möglich im Datenverarbeitungsprozess beibehalten. Erst bei der Auswertung der Daten für die Visualisierung werden die Daten an ihren originären Positionen ausgelesen und ausgewertet.
- Es entsteht kein zusätzlicher Vorverarbeitungs-Berechnungsaufwand, wie ihn etwa eine Konvertierung der Daten vor deren Visualisierung in ein anderes Format mit sich bringen würde.

Die wichtigsten Nachteile dieses Ansatzes:

- Es ist teilweise sehr aufwendig, die Visualisierungsalgorithmen an die vielen verschiedenen Gitterstrukturen anzupassen. Wird eine neue Visualisierungsmethode in das System integriert, muss für jeden unterstützten Gittertyp getrennt eine Algorithmik entwickelt werden. Umgekehrt: Wird ein neuer Gittertyp in das System integriert, müssen alle bestehenden Visualisierungsmethoden entsprechend angepaßt bzw. erweitert werden.
- Dies kann man teilweise umgehen, indem man die Funktionen, die die Daten aus dem Gitter auslesen, abstrahiert, die Gitter sozusagen hinter intelligentere Funktionen kapselt. Die Visualisierungsmethoden, die Werte aus dem Gitter auslesen wollen, bedienen sich hierbei abstrahierender Auslesefunktionen, anstatt selbst direkt auf die Gitterwerte zuzugreifen. Eventuelle Interpolationen der Werte aus umliegenden Nachbarzellen werden dabei im Hintergrund „transparent“ ausgeführt. Der Nachteil bei diesem Ansatz ist, dass diese Interpolationen zur Lauftzeit der Visualisierung wiederholt ausgeführt werden und somit Prozessorzeit kosten.
- Durch die Optimierung einer speziellen Visualisierung auf einen bestimmten Gittertyp, werden gewissermassen „Einmaloptimierungen“ erzeugt. Nur in diesem Spezialfall (spezieller Visualisierungstyp und spezieller Gittertyp) ist der Verarbeitungsprozess optimiert. Bei einer anderen Kombination von Visualisierung und Gittertyp muss erneut Optimierungsaufwand betrieben werden.
- Spätestens bei der Darstellung der Visualisierungsdaten auf der Graphikkarte geht die angestrebte Genauigkeit aufgrund von limitierter Auflösung, Texturgröße, Speicher, Interpolation, etc. zu einem gewissen Teil verloren.
- Die „Genauigkeit“ im Originaldatensatz ist ohnehin nur an die Gitterpunkte / Kanten / Zellen gebunden. Sitzen die Werte beispielsweise auf den Ecken einer Volumenzelle und der Wert in deren Mitte wird benötigt, so muss aus diesen Randwertern interpoliert werden. Hier geht unvermeidbar „Exaktheit“ verloren, da Strömungssimulationen heutzutage nicht mit kontinuierlichen, sondern mit diskretisierten Raumbereichen bzw. Volumen arbeiten. Man muss also für jeden Gittertyp gesonderte Interpolationsfunktionen implementieren.
- Durch die Auswertung der verschiedenen Gitterstrukturen zur Laufzeit geht wertvolle Prozessorzeit verloren, die z.B. für Nachbarschaftssuche (unstrukturierte, unsortierte Gitter) oder Datenbaumtraversierung verwendet werden muss.

Viele Arbeiten in diesem Bereich konzentrieren sich darauf, das Auslesen der Daten in ihren speziellen Formaten schnell und effizient zu realisieren. Es existieren viele hoch-

spezialisierte Ansätze, die im Einzelfall immer versuchen, unter Beibehaltung der original Datengitterstrukturen, die Visualisierung sehr effizient zu machen. Wie eingangs beschrieben, hängt diese Optimierung aber sehr stark von den verwendeten Gittertypen und den darauf aufsetzenden Visualisierungsmethoden ab. Es werden hierbei immer „Einmaloptimierungen“ generiert. Man ist immer gezwungen, für unterschiedliche Kombinationen von Datentypen und Darstellungsmethoden jeweils andere Optimierungsansätze zu bestimmen (z.B. [BG05, BGS04, Ben05]) und neu zu implementieren.

**Konvertierung nach der Visualisierungsberechnung**

Aus der Erkenntnis, dass spätestens bei der Abfrage von Werten, die nicht direkt auf der Struktur des Originalgitters liegen, Interpolationen nötig werden, die die Genauigkeit der Ergebnisse beeinflussen und der Tatsache, dass auch die Darstellung auf der Graphikkarte an Auflösungsbegrenzungen, Speicherlimits und Interpolationen gebunden ist, stützt sich dieser Ansatz. Die Daten bzw. deren Speicherung werden bereits vor deren Auswertung durch die Visualisierung hinsichtlich Zugriffsgeschwindigkeit und Effizienz optimiert. Die Simulationsdaten werden vor dem Start der Visualisierung von ihrem ursprünglichen Format in eine für die Darstellung optimierte Struktur konvertiert.

Auch dieser Ansatz hat gewisse Nachteile:

- Die Genauigkeit der Eingangsdaten wird durch die zusätzlich eingeschobene Konvertierung gemindert. Anstatt die Interpolationen auf den original Gitterwerten auszuführen, werden diese nun mit Hilfe des Konvertierungsgitters berechnet. An dieser Stelle ist es also sehr wichtig, dass die Werte, die aus einem Konvertierungsgitter ausgelessen werden, durch entsprechende Fehlerabschätzungen und -Messungen hinreichend genau an die Originaldaten herankommen. Hier muss man bei der Konstruktion der Konvertierung entsprechende Vorkehrungen treffen (vgl. Abschnitt 5.6.4), um z.B. auszuschliessen, dass dabei kritische Stellen wie Peaks oder Werteschwellen unabsichtlich „geglättet“ werden oder gar ganz aus der Datenstruktur verschwinden.
- Die Konvertierung der Daten benötigt Zeit. Diese Zeit kann man allerdings in einem Pre-Prozess aufwenden, d.h. einmalig bevor die Visualisierung gestartet wird. Anschliessend kann man immer mit den konvertierten Daten arbeiten und muss sie nicht jedesmal erneut umwandeln. Bei „Online“-Visualisierungen, die also ihre Daten erst während der Laufzeit erhalten (z.B. Stück für Stück, immer direkt dann, wenn die Simulation einen neuen Datensatz errechnet hat), kann man diese Konvertierungszeit allerdings nicht in einem Vorverarbeitungsprozess abarbeiten. Dennoch profitiert auch hier die Visualisierung von der Konvertierung, da man dann bei der Darstellung ohne Zusatzaufwand viele unterschiedliche Visualisierungsmethoden auf die eingegangenen (und konvertierten) Daten anwenden kann, ohne dass jede Visualisierung erneut die Daten auf spezielle Weise auswerten muss.

Jedoch bietet diese Vorgehensweise eine Reihe von Vorteilen:

- Für jede Visualisierungsmethode ist nur ein Modell des Datenzugriffs nötig. Dies

entspricht der Idee, zwischen die eigentlichen Daten und die Visualisierungsmethoden eine Abstraktionsschicht zu legen, die die nötige Interpolationsfunktionalität enthält und transparent wegkapselt. Es entsteht der Vorteil, dass diese Interpolation nicht zur Laufzeit der Visualisierung und auch nicht auf unterschiedlichen Gittertypen arbeiten muss.

- Durch den vereinfachen Datenzugriff steht insgesamt mehr Prozessorzeit für den Rest der Applikation und damit für die Visualisierung als solche zur Verfügung.
- Das System kann schnell um weitere (neue) Visualisierungsmethoden ergänzt werden. Diese müssen ihre Datenlese-Routinen nicht an viele unterschiedliche Datenstrukturen, d.h. Gitter, anpassen, sondern die Methode des Datenzugriff ist zusammen mit deren Struktur vereinheitlicht. Es muss also nur eine einzige Methode des Datenzugriffs in der jeweiligen Visualisierungsmethode realisiert werden.
- Soll ein neuer Gitterdatentyp visualisiert werden, muss lediglich die entsprechende Stelle im Konvertierungsschritt angepasst werden, nicht jedoch sämtliche Visualisierungsmethoden.
- Durch die Entkopplung von Datenhaltung und Visualisierung sind diese getrennt voneinander optimierbar. Die Visualisierung ist nicht mehr direkt an den Aufbau der Datenstrukturen der Simulation gebunden, sondern kann losgelöst von ihr entwickelt werden. Es gibt keine direkten Abhängigkeiten mehr.
- In der Praxis stehen bei der Visualisierung von großen Strömungsdatenmengen die großen komplexen Strömungszusammenhänge im Vordergrund. Kleine präzise Details, wie etwa lokal entstehende Mikroturbulenzen sind eher von sekundärem Interesse. Diese Idee verfolgt der Ansatz, die Eingangsgitterstrukturen durch ein adaptives Visualisierungsgitter zu erfassen. Hier ist weiterhin gewährleistet, dass charakteristische Phänomene (wie z.B. lokale Wertspitzen oder Schwellwerte) erfasst werden, jedoch wird gleichzeitig der Datenzugriff durch die vereinheitlichte Struktur optimiert.
- Durch die intelligente Integration von Glättungs-, Sortierungs- und Fehlermeßwertverfahren, ergeben sich neue Ansätze zur Kompression der Daten und deren Überführung in ein progressives Format.

Insgesamt gesehen verspricht die Verwendung des zweiten hier vorgestellten Ansatzes die Entwicklung ein flexibleren und effizienteren Visualisierungssystems, wie es der Idee dieser Arbeit entspricht. Ein proprietäres Datenformat wird entwickelt, welches für die Visualisierung optimiert ist. Es integriert die gewünschten Eigenschaften und gewährleistet die Abkopplung von den, für den reinen Speichervorgang optimierten Simulations-Datenformaten für Strömungsdaten (Kapitel 4.2.10).

Damit läßt sich der Aufwand, den die Visualisierung normalerweise zur Laufzeit hat, in einen vorgeschalteten Prozess (siehe Abbildung 5.20) auslagern: Die Datenmenge wird vor der eigentlichen Visualisierung vorbearbeitet. Hier findet bereits eine Sortierung und Analyse der Daten statt. „Unwichtige“ Teile werden in der Prioriät nach hinten gelegt, „wichtige“ nach vorne. Die Idee des progressiven Formates ist es nun, zunächst nur die „wichtigen“ Teile zu laden und je nach Möglichkeit die „unwichtigen“ im Anschluß. So wird sichergestellt, dass

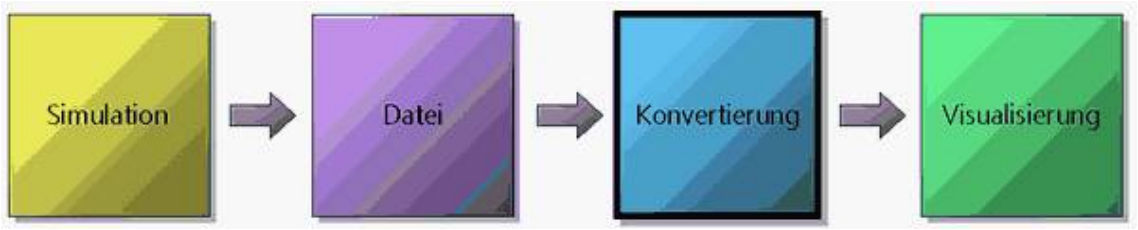



***Abbildung 5.20:*** *Vorgeschaltete Konvertierung: Vor dem Start der eigentlichen Visualisierung werden die ursprünglichen Strömungsdaten in ein neues (progressives) Format gewandelt, das für die Darstellung optimiert ist.*

auch dann die wesentlichen Ausprägungen im Strömungsdatensatz dargestellt werden, wenn auch nur ein kleiner Teil der Datenmenge in den Speicher geladen werden kann. Zusätzlich werden diese Teile so abgelegt, dass es möglich ist, eine räumliche Priorisierung zuzulassen. In der Praxis läßt sich das dann dafür ausnutzen, dass zum einen nur Teile geladen werden, die von Proben abgedeckt werden, zum anderen, nur Teile, die direkt um den Betrachter der Szene herum liegen. Hier wird das angesprochene Data Clipping (vgl. Kapitel 5.2.1.1) unterstützt.

### 5.6.1 Überblick

Die hier entwickelten progressiven Gitter sind eine spezielle Form der hierarchisch strukturierten Gitter. Hierarchisch strukturierte Gitter sind im Gebiet der Computer Graphik gut bekannt und weit verbreitet [WvG92, Sam84]. In ihrem eigentlichen Kontext werden sie verwendet, um räumlich große Mengen von Geometrie-Fragmenten zu arrangieren. Sie werden so angelegt, dass für jede gegebene Zelle bestimmbar ist, welche Fragmente sie enthält. Diese Gitter sind adaptiv, da die verwendeten Zellen genau dann in weitere Unterzellen zerlegt werden, wenn die eingeschlosse Datenmenge einen bestimmten Schwellwert überschreitet.

Die progressiven Gitter verwenden Technologien und Ideen aus progressiven Datenformaten, die eng mit dem Thema Digitale Bildverarbeitung verwandt sind (Wavelet, JPEG-Kompression) [FL99, CMPS97, ZMFM97]. In diesen Formaten werden die Daten nach ihrem Level of Detail sortiert gespeichert. Auf diese Weise kann man leicht Daten bezogen auf eine gewünschte Auflösung aus der Datei einlesen. Als Ergebnis kann man die Menge der Daten, die übertragen und/oder bearbeitet werden muss, steuern und dadurch anpassen; beispielsweise an veränderlichen Systemresourcen oder Benutzeranforderungen. Geometrie Daten können z.B. übertragen und dargestellt werden, während ihre Granularität gleichzeitig immer weiter verfeinert wird. So erhält der Betrachter schnell einen Eindruck von der Szene oder dem übertragenen Bild bereits kurz nach dem Start der Übertragung [Hop96, Sah03].

Überträgt man nun diese progressive Idee auf CFD Simulationsdaten, kann man auch hier von diesen Vorteilen profitieren. Die Daten werden in Bezug auf die enthaltenen Informationen sortiert, beginnend mit den „wichtigsten“. Für diese Aufgabe wird ein Verfahren zur räumlichen Aufteilung der Daten entwickelt, das - basierend auf übetragenen Ideen aus dem Geometrie und Bildverarbeitungsbereich - die Grundlage für das progressive Format bildet.

Das Prinzip der adativen Gitter Verfeinerung (*adaptive mesh refinement* - AMR) stammt aus dem Gebiet der numerischen Berechnungen [BD99] für die Lösung von partiellen diffentialgleichungen mit Finiten Differenzen. Die AMR Methode [BO84] basiert auf der Idee rekursiver lokaler Verfeinerungen eines gegebenen groben Startgitters. Die Verfeinerungen werden hiebei von einem Fehler gesteuert, der von einer entsprechenden Fehler-Metrik bestimmt wird. Die Idee dieser Methode ist es, numerische Berechnungen nur dann durchzuführen, wenn sie nötig sind.

Außer im Gebiet der numerischen Analyse und wissenschaftlichen Berechnung gibt es nur wenige Ansätze in Bezug auf die technisch-wissenschaftliche Visualisierung mit Hilfe von progressiven Datenformaten. In der Arbeit von Holliday et al. [HN00] wird eine Methode vorgestellt, die ein rechtwinkliges Gitter in Tedraer und Oktaeder zerlegt. Die ursprünglichen Gitterwerte werden mit Hilfe sogenannter *Coon Patches* angenähert. Falls der resultierende Fehler von der Ausgangszerlegung zu groß ist, wird die Partionierung lokal weiter verfeinert. [WKL+01] verwendet zwar hierachisch strukturierte rechtwinklige Gitter, konzentriert sich aber auf das darauf aufsetzende Interpolationsverfahren, das Diskontinuitäten an den Rändern in Bezug auf unterschiedliche Unterteilungsebenen vermeiden soll.

Ein weiterer Ansatz zur Erzeugung von hierarchischen Gittern wird in [GLE97] beschrieben. Hier bilden die Autoren die Hierarchie ausgehend von dem gröbsten Level bzw. gröbsten Gitter und das damit verbundene Approximationsproblem wird in den Finiten Elemente Raum übertragen. Die Eingabedaten werden als Diskretisierung einer glatten Funktion betrachtet, die durch adaptive Gitterverfeinerung und Fehlerkontrolle angenähert wird. Ausgegangen wird hiebei von einer diskreten Funktion, die auf einem groben Gitter definiert ist. In [GG98] wird der damit verbundene adaptive Gitter Algorithmus vorgestellt, der auf einer adaptiven und regulären Verfeinerung eines groben Startgitters besteht, dass konsistent in Tetraeder und Oktaeder zerlegt wird.

Passende adaptive Visualisierungsmethoden werden beispielsweise in [OR97, NORS97] vorgestellt. Z.B. wird in einem Iso-Flächen Algorithmus ein Gleichgewicht zwischen Korrektheit und Geschwindigkeit erreicht, indem ein Initialwert für den *visuellen Fehler* gesetzt wird. Eine hierarchische Repräsentation von Vektorfeldern wird in [HWHJ99] beschrieben. Zuerst wird das Vektorfeld in disjunkte Cluster geteilt. Die Vektoren in jedem Cluster werden mit einem einzigen Durchschnittsvektor angenähert. Für einen gegebenen Punkt in diesem Feld werden zwei Strömungslinie mit Hilfe des Runge-Kutta Verfahrens errechnet. Für die erste Linie werden hierbei die Originaldaten verwendet. Die zweite Linie entsteht auf dem approximierenden Gitter. Ist die Abweichung zwischen beiden Linien innerhalb dieses Clusters größer als ein gegebener Schwellwert, wird der Cluster weiter zerlegt. Auf diese Weise entstehen hierachische BSP-Bäume, die speichersparend sind, da sie ohne Gitter auskommmen. In [GPR+01] wird ein phsyikalisches Clustering Modell vorgestellt, das sog. *Cahn-Hiliead-Model*. Es beschreibt die Auftrennungs-Phase, so daß ein kontinuierliches multiskalen Clustering auf Vektor Feldern entsteht.

### 5.6.2 Grundidee

Die Grundlage für die progressiven Gitter und Eingabedaten für den Konvertierungs-Algorithmus ist eine reguläre Diskretisierung des Datenraums. Hierbei ist wählbar, ob diese Diskretisierung aus der original Simulation kommen soll oder aus einem regulären Abtastgitter eines irregulären / unstrukturierten Gitters ermittelt werden muss. Die progressiven Gitter werden hierbei *generisch* angelegt, d.h. in eine Konvertierung-Umgebung integriert. Dadurch wird es möglich, die topologische Struktur jederzeit beliebig auszutauschen: Die im progressiven Gitter verwendeten Zell Typen und räumlichen Unterteilungs Schemata sind wählbar (Abbildung 5.21). Diese definieren die Topologie des erzeugten progressiven Gitters und wie die Gitterdaten im Nachhinein interpretiert werden müssen. Dasselbe gilt für die darauf aufbauenden Interpolationsalgorithmen und die Fehlerabschätzung, über die die Abweichung vom Eingabegitter berechnet wird. Die verwendeten Zellen werden in einer hierarchichen Struktur abgelegt. Der auszulesende numerische Wert ergibt sich dann aus der entsprechenden Zelle bzw. deren Randpunkten. Jede Zelle beinhaltet eine bestimmte Anzahl von Eckpunkten und eine bestimmte Zahl von weiteren Kinderzellen, deren Daten ggf. mit in die Berechnungen einfließen, um die Genauigkeit zu steigern.

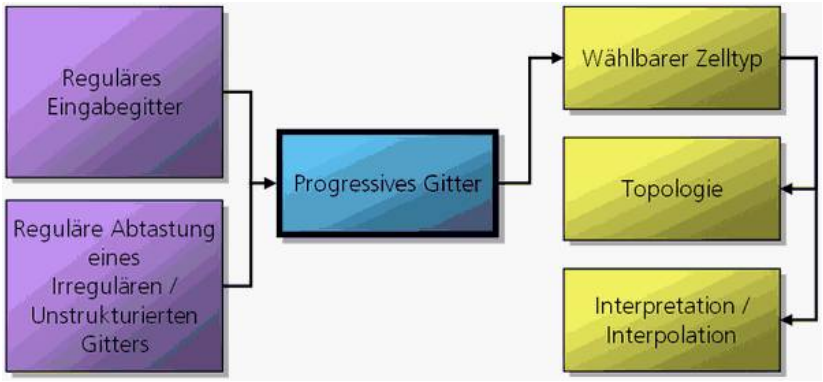



***Abbildung 5.21:*** *Die Konvertierung und das dabei entstehende progressive Gitter.*

Dieser allgemeine Ansatz ermöglicht es, die Zeit als weitere Dimension in die Progressiven Gitter zu integrieren und dadurch einen 4 dimensionalen Gittertyp zu konstruieren. Auf diese Weise kann man auch zeitliche Kohärenzen mit für die Kompression aussnutzen. Diese Grundidee findet man heutzutage beispielsweise in vielen Videodatenformaten wie MPEG [11192] oder DivX [DIV]. Auch dort werden zeitliche Kohärenzen ausgenutzt, um die Kompressionsraten zu steigern, denn streng genommen lassen sich auch Videos als zeitlich veränderliche 2D Skalar-/Vektorfelder betrachten.

### 5.6.3 Die Gitterzellen

Die *Zelle* ist eine elementarer Bestandteil der Progressiven Gitter. Allgemein betrachtet speichert sie:

- Verweise auf ihre Eckpunkte
- Informationen darüber ob und inwiefern die Zelle unterteilt ist
- Verweise auf ihre „Mutter“ und ihre „Kinder“-Zellen (falls vorhanden)
- Numerische Informationen ihren Inhalt betreffend
- Verweise auf eine bestimmte Anzahl von Nachbarzellen (falls nötig)

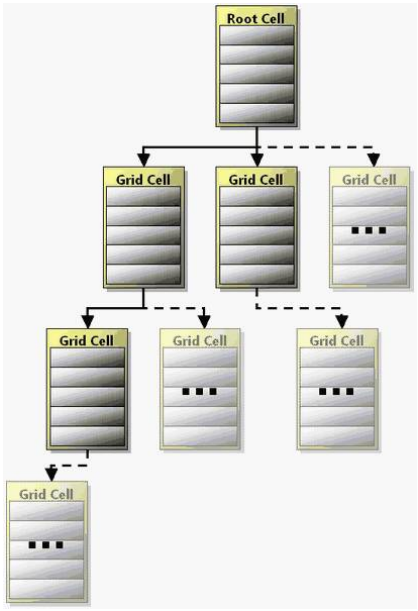



***Abbildung 5.22:*** *Hierarchie der Progressiven Gitter: Die Zellen werden in Form eines gerichteten azyklischen Graphens im Speicher abgelegt.*

Diese Speicherstuktur kann als gerichteter azyklischer Graph interpretiert werden (siehe Abbildung 5.22). Falls möglich, werden die Werte für die Eckpunkte der Zelle direkt aus den Werten der regulären Eingabegitter übernommen. Das progressive Gitter wird nicht feiner aufgelöst als das Eingabegitter, da dadurch keine weiteren bzw. genaueren Informationen generiert würden. Die Punkte, die im progressiven Gitter gespeichert werden, enthalten ihre Koordinaten in beliebiger Dimension, wenngleich an dieser Stelle 4 dimensionale Werte vorgesehen sind. Diese Koordinaten legen die Position der Zelle im Zeit-/Volumen Datenraum fest. Zusätzlich speichern die Eckpunkte noch die eigentlichen Datenwerte (also Skalar oder Vektordaten), die an dieser Stelle aus dem Eingabegitter ausgelesen wurden. Ein weiterer Parameter wird integriert, der festlegt, ob der betroffene Punkt außerhalb oder innerhalb des Eingabegitters liegt (bezeichnet als *inoutflag*). Er wird benötigt, da sich durch die Gestalt der gewählten Zellen nicht unbedingt eine 1:1 Überdeckung des Originalgitters erreichen läßt, falls dieses z.B. beliebige Struktur hat, die progressiven Zelltypen aber rechtwinklig gewählt sind. So können Zellen entstehen, bei denen manche Eckpunkte außerhalb des Originalgitters liegen. Diese werden entsprechend markiert (Abbildung 5.23). Der Grund dafür, dass eine Zelle lediglich Verweise auf ihre Eckpunkte speichert anstatt die Punkte selbst, liegt darin,

dass diese Punkte auch von angrenzenden Zellen verwendet werden. Um den Speicherbedarf und die Verwaltung der Punkte klein zu halten, werden nur die Referenzen auf die Punkte gespeichert. Eine Standard-Zelle ist in Bild 5.24 zu sehen.

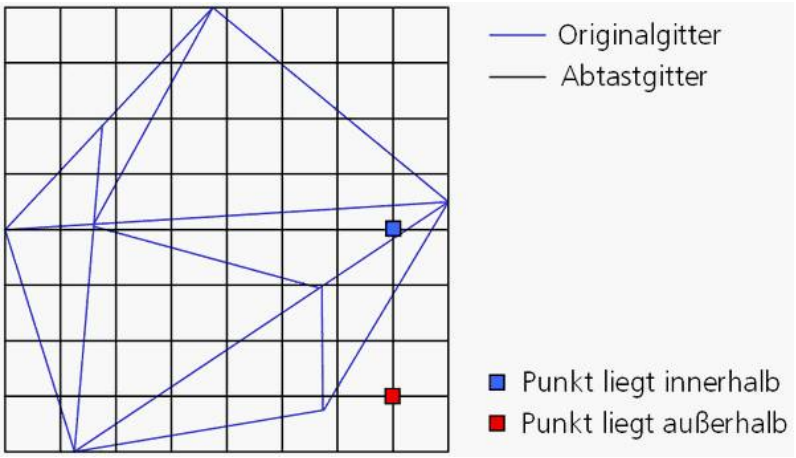



***Abbildung 5.23:*** *Orignal- und Abtastgitter: Aufrund ihrer unterschiedlichen Struktur und Zelltypen, kann es vorkommen, dass Eckpunkte des Abtastgitters (regulär und orthogonal) außerhalb des Originalgitters (unstrukturiert und irregulär) liegen. Die Punkte werden im progressiven Gitter entsprechend markiert.*

Die wichtigste numerische Information für den Konvertierungsalgorithmus, die mit einer Zelle zusammenhängt, ist der Approximationsfehler. Er beschreibt die Approximationsqualität in Bezug auf das Originalgitter. Zusätzlich besteht die Möglichkeit, noch einige informative Werte zu speichern (z.B. der kleinste, der größte und der durchschnittliche Wert, den die Daten innerhalb der Zelle annehmen).

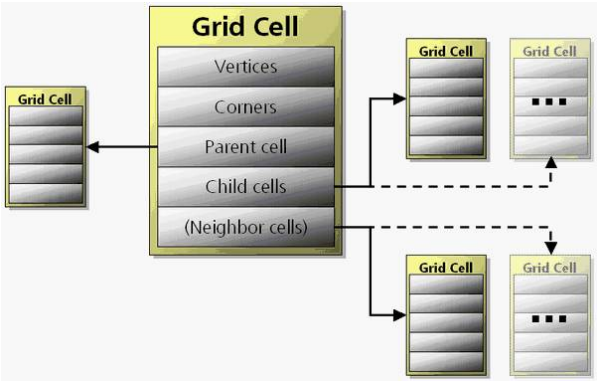



***Abbildung 5.24:*** *Aufbau einer Zelle des Progressiven Gritters*

### 5.6.4 Das Konvertierungsschema

In diesem Abschnitt wird beschrieben, auf welche Art und Weise aus den Eingabedaten die progressiven Gitter in die progressive Datenstruktur umgewandelt werden.

#### 5.6.4.1 Die Initialpartionierung

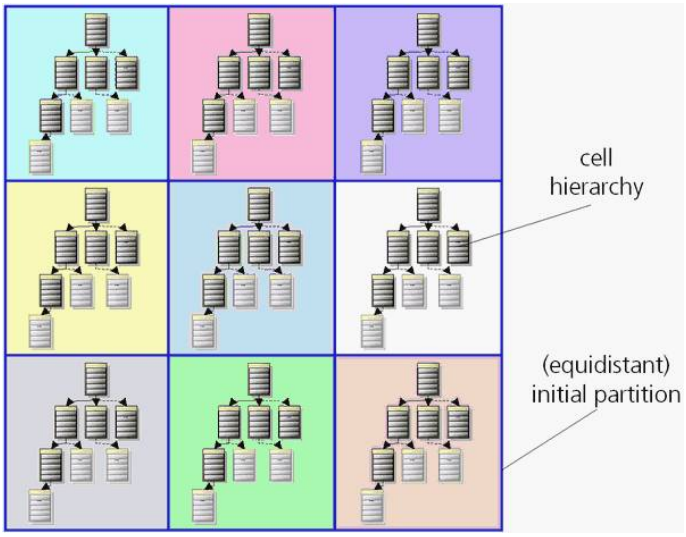



***Abbildung 5.25:*** *Initial Partitionierung*

Die Hauptaufgabe während der Konvertierung besteht darin, die Zellenhierarchie aufzubauen. Das ist der zeitaufwendigste Teil. Zusätzlich muss der Konverter festlegen, in welcher Reihenfolge die Zellen-Informationen abgespeichert werden. Die Konvertierung startet mit einer Initialpartitionierung (siehe Abbildung 5.25). Im einfachsten Fall sind das rechteckige Zellen, deren zugehörige Gitteranordnung direkt aus der Bounding Box des Simulationsgitters bestimmt werden kann. Im Falle einer Dekomposition in Tetraeder besteht die Initial Partitionierung aus einer Zerlegung der Bounding Box in eine gegebene Anzahl von Tetraeder. Die Gestalt der Zellen der Initialpartitionierung ist hierbei unabhängig von deren Anzahl, d.h. man kann beides getrennt voneinander festlegen. Die Zellen der Initialpartitionierung werden als *Wurzelzellen* oder *root cells* bezeichnet. Die progressiven Gitter können mehr als eine Wurzelzelle enthalten. Zusätzlich wird eine Funktion benötigt, die zu einem gegebenen Punkt die zugehörige Wurzelzelle findet. Die Idee eine Initialpartitionierung mit verschiedenen Wurzelzellen einzuführen, auf denen dann jeweils progressive Daten fußen, ist einfach: Die Konvertierung kann parallel implementiert werden, ohne dass Overheads bzw. Latenzen durch Synchronisation entstehen: Jede Wurzelzelle kann unabhängig von den anderen Wurzelzellen in progressiv organisierte Zellen zerlegt werden. Ein weiterer sehr wichtiger Vorteil ist der, dass man durch die Initialpartionierung in der Lage ist, für jede Wurzelzelle getrennt festzulegen, welche Genauigkeit beim Traversieren der darauf aufbauenden progressiven Struktur erreicht werden soll. Für Szenen, die größer als der Sichtbereich des Betrachters sind, ermöglicht diese Vorgehensweise eine Blickpunkt abhängige Genauigkeit pro Wurzelzelle einzuführen.

Auf jeder Wurzelzelle der Initialpartitionierung wird also eine Hierachrie von progressiven Gitterzellen erzeugt. Innerhalb einer Zelle werden auszulesende Daten durch Interpolation ermittelt. Um also an die konkreten Daten eines Punktes im Raum zu gelangen, wird zunächst die zugehörige Zelle in der Initialpartionierung bestimmt. Anschließend wird die darauf auf-

setzende progressive Datenstruktur bis zu einer vorgegebenen Fehlerschranke durchlaufen. Die entsprechende Zelle des progressiven Gitters wird identifiziert und der auszulesende Wert aus deren Randpunkten interpoliert.

#### 5.6.4.2 Bestimmen der Approximationsgenauigkeit

Für jede unpartitionierte Zelle wird ein Approximationsfehler ermittelt, indem man Abtastpunkte bestimmt, die innerhalb der Zelle liegen. Die Werte der gespeicherten physikalischen Größe (z.B. Temperatur) für diese Punkte werden durch Interpolation innerhalb der Zelle ermittelt und mit den Werten aus dem ursprünglichen Eingabegitter verglichen. Diese Vorgehensweise erlaubt, die Topology der progressiven Gitter unabhängig von den Eingabedaten zu erzeugen. Allerdings hat diese Unabhängigkeit auch einen kritischen Punkt: Da keine Originalwerte des Eingabegitters im progressiven Gitter zu finden sind, ist die geschickte Wahl der Abtastpunkte entscheidend für die Qualität der Konvertierung. Beschränkt man sich hier beispielsweise lediglich auf die Punkte des Originalgitters zur Festlegung der Abtastpunkte, so können kritische Artefakte entstehen, falls keiner dieser Punkte innerhalb der zu erzeugenden progressiven Gitterzelle liegt, man demzufolge für sie auch keine Approximationsgenauigkeit bestimmen könnte, auch wenn die Kanten des Originalgitters die progressive Zelle schneiden würden. Aus diesem Grund ist es z.B. sinnvoll, statt dessen einfach eine gleichmäßige Abtastung des Originalgitters mit gleichmäßg verteilten Abtastpunkten zu verwenden. Wichtig hierbei ist, dass die Abtastung fein genug ist, um alle wichtigen Details der Eingabedaten zu erfassen, so daß der Vergleich dieser Werte mit denen aus der progressiven Zelle interpolierten, sinnvolle Ergebnisse liefert.

Der *Approximationsfehler* an jedem Abtastpunkt wird definiert als Linearkombination der Fehler jedes Skalar- oder Vektorfeldwertes und dem *inoutflag*. Bei der Konvertierung kann festgelegt werden, welche Skalar- und Vektorfelder mit in die progressive gewandelte Strukur übernommen, d.h. auch dort gespeichert werden sollen und insbesondere welche von ihnen einen Einfluß auf die Approximationsfehlerberechnung haben sollen. So ist es ggf. möglich, bestimmte Datenfelder zur Bestimmung des Fehlers auszublenden oder nur manche der Eingabedatenfelder bei der Konvertierung zu übernehmen (z.B. nur die Temperaturwerte aus seinem Datensatz mit Temperatur, Druck und Dichte). Ferner erlaubt dieses Vorgehen ein, speziell für ein bestimmtes Skalar- oder Vektorfeld optimiertes progressives Gitter zu erzeugen, um dadurch bessere Kompressionsraten zu erreichen. Um jedoch den Aufwand dieser Vorgehensweise (getrennte progressive Gitter pro Datenfeld) zu vermeiden, werden Gewichte für die einzelnen Fehler mit in die Linearkombination des Approximationsfehlers integriert. Falls ein bestimmtes Datenfeld so einen größeren Einfluß bei der Bestimmung des Approximatinsfehlers haben soll, wird sein Gewicht entsprechend höher gesetzt. Sollen die Ränder des Originalgitters einen stärkeren Einfluß haben, so wird der Koeffizient für das *inoutflag* vergrößert.

#### 5.6.4.3 Erzeugen der Hierarchie / Subdivision

Der Konverter teilt die vorhandenen Zellen so lange, wie ihr Approximationsfehler größer als eine vorgegebene Schranke ist oder eine vorgegebene Auflösung erreicht ist. Hierbei wird

immer die Zelle mit dem größten Fehler zuerst geteilt (Abbildung 5.26). Daraus entstehen neue Kinderzellen, die erwartungsgemäß einen kleineren Approximationsfehler aufweisen, wie ihre Ursprungszellen. Durch diese Vorgehensweise wird garantiert, dass der maximale Fehler in den Blätterzellen des entstehenden Graphen (Baumes) sehr schnell schrumpft. Die neu entstehenden Kinderzellen werden in derselben Reihenfolge gespeichert, in der ihre Eltern vorliegen. So wird es möglich, das progressive Gitter nur bis zu einer bestimmten Fehlerschwelle in den Speicher zu laden. Die zugehörigen Zellen werden als Block ausgelesen.

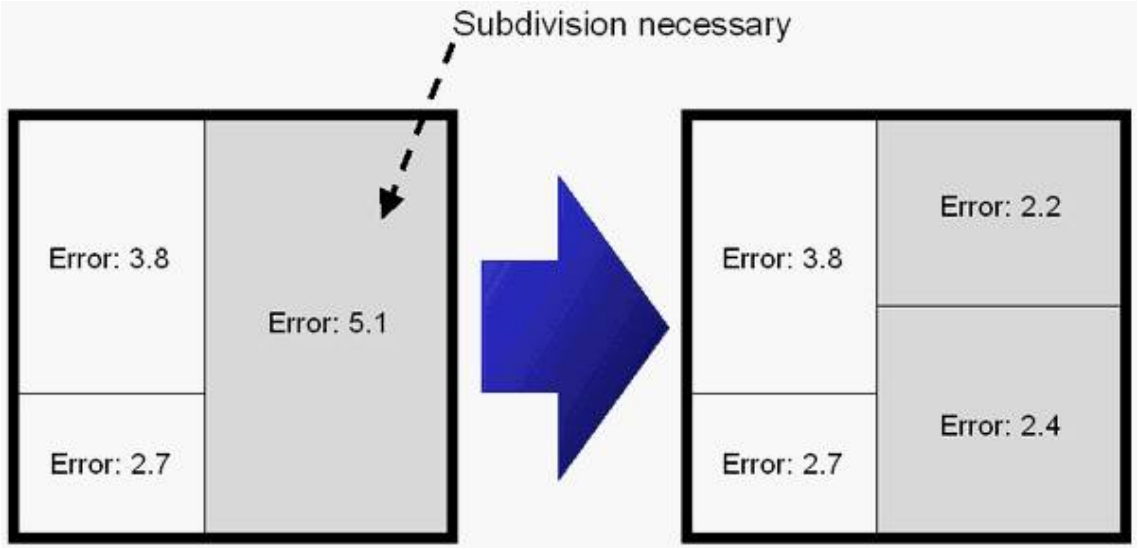



***Abbildung 5.26:*** *Subdivision bei progressiven Gittern: Die Zellen mit dem größten Approximationsfehler werden zuerst unterteilt.*

Bereiche des Originaldatensatzes mit niedrigen Werte-Frequenzen, d.h. geringen Schwankungen zwischen den gespeicherten Werten, werden durch wenige Unterteilungen (und damit weniger Zellen) recht schnell angenähert. Bereiche mit hohen Schwankungen in den Daten benötigen entsprechend mehr Unterteilungen. Da jeder Strömungsdatensatz Bereiche hat, in denen es relativ konstante Werte gibt, kann das auf diese Weise für die Kompression der Daten und damit für die erzeugte Hierarchie der Zellen ausgenutzt werden. Weitere Details hierzu befinden sich in [MSSL03a].

#### 5.6.4.4 Zelltypen

Je nachdem, welcher Zelltyp für die progressiven Gitter gewählt wird, ändert sich auch die Gestalt der erzeugten progressiven Hierarchie. Um den Umgang mit verschiedenen Zelltypen zu vereinfachen, werden die entsprechenden zugehörigen Funktionen in abgetrennte Module ausgelagert, so daß sie leicht gegen andere ausgetauscht werden können. Hierzu zählen beispielsweise die Funktionen für die Interpolation oder für die Berechnung des Approximationsfehlers der Zelle.

Zwei typische Zelltypen werden an dieser Stelle einmal genauer untersucht:

**Rechtecke**

Rechtecke sind einfach zu handhaben. Ihre Ecken können direkt mit Punkten aus dem Abtastgitter assoziiert werden. Die zugehörigen Tests und Interpolationsschemas sind einfach und effizient, was der Leistungsfähigkeit des darauf aufsetzenden Visualisierungssystems zugute kommt (Abbildung 5.27).

**Tetraeder**

Tetraeder erlauben eine bessere und schnellere Approximation der Originaldaten bei niedrigerer Zellunterteilungstiefe, da ihre Zellwände nicht in festen Winkeln zu den Raumachsen stehen müssen. Die Kompressionsrate ist demzufolge hierbei besser als bei rechteckigen Zelltypen. Allerdings ist es hier selten gegeben, dass die Eckpunkte der Tetraeder genau die Punkte des Abtastgitters treffen. Um trotzdem die entsprechenden Werte zu ermitteln sind mehr Berechnungen nötig. Die Konstruktion der Gitter ist aufwendiger. Um zu verhindern, dass das progressive Gitter Zellen enthält, die zu klein werden und es degeneriert, sollte dafür gesorgt werden, dass jede Tetraederzelle zumindest einen Punkt des Abtastgitters enthält.

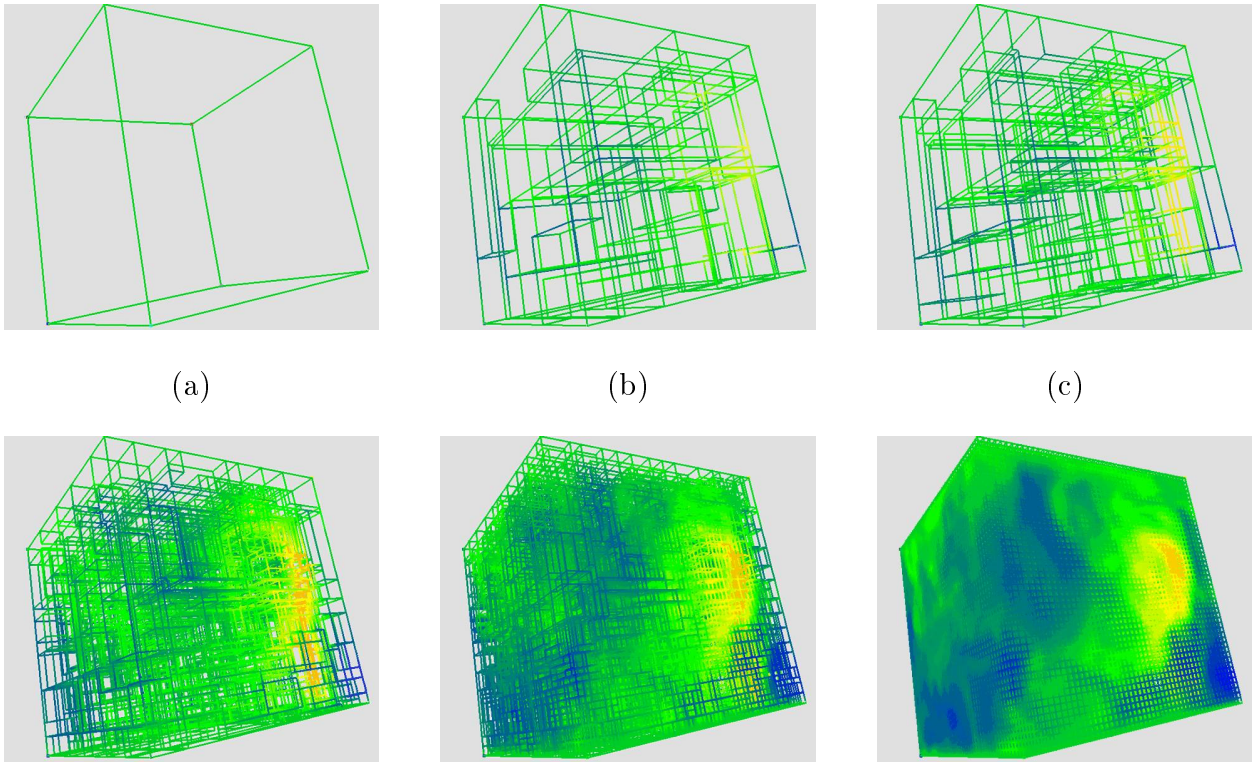



***Abbildung 5.27:*** *Verschiedene Auflösungen desselben progressiven Gitters: (a) Gitterstruktur bei Hierarchielevel 0 bzw. mit 1 Zelle. (b) mit 50 Zellen. (c) mit 100 Zellen. (d) mit 1000 Zellen. (e) mit 10000 Zellen. (f) mit der vollen Auflösung, d.h. 110286 Zellen. Die Kanten im Beispiel sind entsprechend der zugrundeliegenden Skalarwerte gefärbt.*

#### 5.6.4.5 Der adaptive kD-Baum

Der kD-Baum [Ben75, Ben79]) ist ein spezielles Raumunterteilungsschema, für die assoziative Suche innerhalb eines k-dimensionalen Raumes. Er stammt aus einer speziellen Untermenge der Binary Space Partitioning (BSP) Bäume [SBGS69], da er auf demselben Prinzip basiert und lediglich die Wahl der raumaufteilenden Elemente auf achselparalle Hyperebenen festlegt. Beschränkt man die Zellgestalt der progressiven Gitter zunächst auf kartesische Rechtecke, kann man im Gegensatz zum Octree Schema [Gla84] kontrollieren, entlang welcher Achse die nächste Stufe der Verfeinerung durch Unterteilung stattfinden soll. Die Daten für die vier neu entstehenden Eckpunkte, die die Unterteilungs Hyperebene aufspannen, haben einen großen Einfluß auf die Gesamtqualität der Partitionierung. Aus diesem Grund wird die Zelle nicht in zwei Kinderzellen gleicher Größe gespalten, sondern die Unterteilungsgrenze unter Einbeziehung der Originaldaten festgelegt. Im einfachsten Fall funktioniert das so, dass der Algorithmus für alle möglichen Partitionierungsebenen, die durch Punkte auf dem Abtastgitter entstehen, den entstehenden Approximationsfehler berechnet. Diese Fehler werden verglichen und schließlich die Hyperebene durch die vier Punkte gelegt, die die kleinsten Fehler erzeugen.

Die herkömmliche bi- oder trilineare Interpolation für 2D oder 3D Daten stellt die einfachste Art und Weise dar, ein kontinuierliches Skalar- oder Vektorfeld aus einem gegebenen diskretisierten Feld anzunähern. Im Detail bedeutet das, dass die Qualität dieser Approximation vom Gradientenfeld abhängt. Die Interpolation ist exakt sobald das Gradientenfeld innerhalb einer Zelle des Approximationsgitters konstant ist. Hier macht es Sinn, die Partition so zu wählen, dass möglichst viele Eckpunkte des Abtastgitters mit gleichen oder ähnlichen Gradienten in eine Zelle fallen. Je größer die Zelle wird, desto größer wird die Spanne der Gradienten der innerhalb gelegenen Punkte. In einem abstrakten Sinn läßt sich also die Homogenität einer Zelle durch einen Vergleich der Gradienten der enthaltenen Punkte bestimmen. Eine **optimale** Partitionierung zeichnet sich demzufolge dadurch aus, dass sie die Homogenität in beiden auf diese Weise neu entstehenden Tochterzellen maximiert.

Die folgende Formel beschreibt eine Partitionierung im 2D Raum entlang der X-Achse. Sie kann leicht erweitert und auf den 3D Raum - genauer auf beliebige Dimension - übertragen werden. Dadurch, dass ein reguläres Abtastgitter der Domain verwendet wird, werden im Beispiel alle entsprechenden Gitterpunkte über ihre kartesischen Indizes $\vec{v_{i,j}}$ identifiziert. Sei $\vec{g_{i,j}}$ der Gradient eines Eingabe Skalarfeldes am entsprechenden Punkt. Ein einfaches effizientes Maß, um die Homogenität $H$ zwischen Gratienten einer gegebenen rechteckigen Zelle $B$ (festgelegt durch die Indizes $i_{low}, i_{high}, j_{low}, j_{high}$) zu bestimmen, ist die Norm aller betroffenen Gradienten Vektoren:

$$H(B) = \left\| \sum_{i=i_{low}}^{i_{high}} \sum_{j=j_{low}}^{j_{high}} \vec{g_{i,j}} \right\|_2^{\alpha} \tag{5.1}$$

So festgelegt, ist die Homogenität eines bestimmten Bereiches fast proportional zu seiner Größe, falls sich die einzelnen Gradienten ähnlich sind. $\alpha(>1)$ ist hierbei ein „Design-“ bzw. „Steuerparameter“, um größeren Bereichen bei Aufnahme neuer Gradienten bzw. zugehöriger Eckpunkte mehr Gewicht zu verleihen. Die Partitionierung teilt eine Zelle entlang einer festen

Achse in zwei Unterzellen. Nun kann man ein Maß für die Qualität einer Zellenpartitionierung $p$ festlegen: Man berechnet die Summe über die Homongenität der Kinderzellen:

$$H_p(B) = \left\| \sum_{i=i_{low}}^{p} \sum_{j=j_{low}}^{j_{high}} \vec{g_{i,j}} \right\|_2^{\alpha} + \left\| \sum_{i=p+1}^{i_{high}} \sum_{j=j_{low}}^{j_{high}} \vec{g_{i,j}} \right\|_2^{\alpha}, p = (i_{low}+1)...(i_{high}-2) \tag{5.2}$$

Im Gegensatz zur ständigen Bestimmung des Approximationsfehlers für die Partitionierung, kann das Schema der maximalen Homogenität der Kinderzellen durch eine Vorberechnung vereinfacht werden, denn alle Gradienten einer Ebene durch die Eckpunkte im Abtastgitter lassen sich durch einen Vektor $\vec{h}$ beschreiben mit

$$\vec{h} = (h_1, h_2, ..., h_i), h_l = \sum_{j=j_{low}}^{j_{high}} \vec{g_{l,j}} \tag{5.3}$$

Die Partition, die $H_p(B)$ maximiert, wird als Grundlage für die weitere Unterteilung in Kinderzellen genommen.

## 5.7 Gesamtsystem Architektur

Nachdem in den vorangegangenen Kapiteln die einzelnen Teile des Systems genauer vorgestellt wurden, wird an dieser Stelle noch einmal ein Überblick über das gesamte Visualisierungssystem gegeben (Abbildung 5.28).

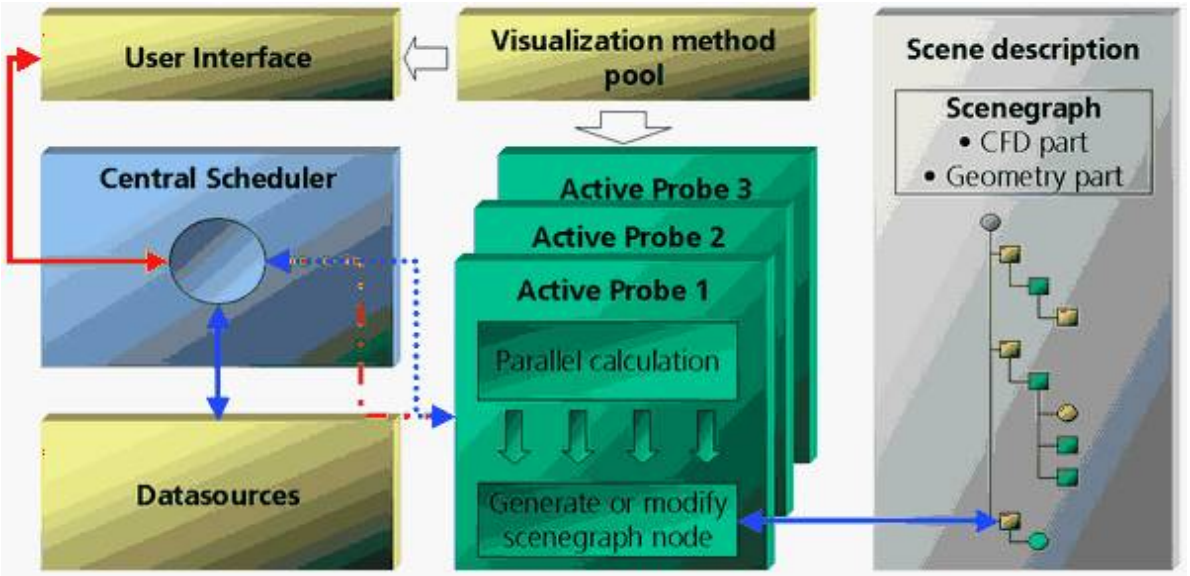



***Abbildung 5.28:*** *Die gesamte Systemarchitektur im Überblick. Einige wichtige Module und ihre Verbindungen sind in diesem schematischen Schaubild dargestellt.*

Das System zerfällt in verschiedene Module, die jeweils unterschiedliche Aufgaben übernehmen [SMS03b, SMS03a]:

**Datenverwaltung**
In diesem Modul des Systems werden eingeladene Daten verwaltet. Hier können die anderen Komponenten des Systems erfragen, welche Daten gerade zur Verfügung stehen. Ferner stellt dieses Modul Informationen zum Typ der Daten bereit (beispielsweise, ob es sich um Skalar oder Vektordaten handelt). Die physikalische Einheit und Bezeichnung des jeweiligen Datenfeldes kann erfragt werden. Außerdem ist hier gespeichert, welcher räumliche und zeitliche Teil der Simulation in welcher Auflösung mit dem jeweiligen Datensatz abgedeckt wird. Es können bei Bedarf bestimmte Teile nachgeladen oder ausgelagert werden. Im Kapitel 5.4 wird auf diesen Bereich genauer eingegangen.

**Visualisierungsmethoden**
Dieses Modul stellt die Informationen über die Visualisierungsmethoden dem System zur Verfügung. Das System kann über eine abstrakte Schnittstelle erfragen, welche Visualisierungsmethoden für welche Art Datenfeld (Skalardaten, Vektordaten) für die Darstellung zur Verfügung stehen. Hierbei werden nur die Visualisierungsmethoden aufgelistet, die auf dem vorgefundenen System lauffähig sind (falls manche von ihnen z.B. das Vorhandensein spezieller Gegebenheiten voraussetzen (z.B. eine konkrete Anzahl CPUs oder bestimmte Graphik Extensions). Jede Visualisierungsmethode ist mit einem Berechnungsteil ausgestattet, der die entsprechende Ausgabegeometrie errechnen kann. Der im Kapitel 5.2.2.1 vorgestellte Visualization Method Pool speichert die zur Laufzeit aktiven Proben.

**Szenenbeschreibung**
Dieses Modul enthält die Szenenbeschreibung für die Ausgabe in Szenegraph Repräsentation. Hier befinden sich alle Daten, die für die graphische Ausgabe benötigt werden. Dazu zählen neben den Parametern für die Ansicht und den Standord des Betrachters auch die Umgebungsgeometrie der Szene, Beleuchtung und schließlich die Geometrie, die durch die aktiven Visualisierungsmethoden erzeugt wird.

**Benutzerinterface**
Dieses Modul stellt die grafische Oberfläche bereit. Neben der Factory für die Panels der einzelnen Visualisierungsmethoden werden hier auch die anderen Komponenten des Systems gesteuert. Die resultierenden Befehle werden dann über den Central Scheduler an die entsprechende Zielkomponente übertragen.

**Central Scheduler**
Der Central Scheduler (siehe Kapitel 5.3.1) dient als zentrale Kommunikationskomponente. Er steuert und koordiniert den Datenaustausch zwischen den Modulen. Ferner startet er nötigenfalls Berechnungen innerhalb der Visualisierungsmethoden und veranlaßt anschließend die nötigen Änderungen in der Szenenbeschreibung.

## 5.8 Zusammenfassung

In diesem Kapitel wird ein Konzept für ein Strömungsdaten-Visualisierungssystem vorgestellt. Das Konzept orientiert sich dabei an den in Kapitel 4 vorgestellten Anforderungen.

Wie in Kapitel 4.2.2 beschrieben, ist ein wesentlicher Grundgedanke bei der Konzeptionierung des Visualisierungssystems für Strömungsdaten der modulare Aufbau. Durch ihn wird es möglich, Programmteile getrennt voneinander zu entwickeln und zu optimieren. Außerdem kann man das System schnell an andere Rahmenbedingungen (z.B. ein bestimmtes Ausgabeformat oder Austautsch der GUI) anpassen (vgl. Kapitel 4.2.3). Eine modularisierte Programmstruktur findet sich heutzutage in vielen Applikationen und ist, ähnlich wie die Design Patterns [Ale01, Mey98], ein gängiges Entwurfsparadigma geworden, das hier Anwendung findet. So wird beispielsweise der Austausch des Ausgabeszenegraphen oder der Benutzeroberfläche möglich.

Durch die Einführung des Proben-Konzeptes und des Visualisization Method Pools wird ein Rahmen für Visualisierungsmethoden geschaffen, der es zum einen erlaubt, mehrere Methoden gleichzeitig zu unterstützen, als auch diese sinnvoll räumlich platzieren und einschränken zu können.

Der Central Scheduler ist der wichtigste Bestandteil des Kernels. Er übernimmt die zentrale Steuerung des Systems bzw. der einzelnen Komponenten und koordiniert die Kommunikation. Über eine abstrahierte Nachrichtenmethode, die sogenannten Actions, wird es möglich, asynchron mit Parametern behaftete Kommandos im System zu verschicken, um die einzelnen Module Daten miteinander austauschen zu lassen.

Anschließend werden die unterschiedlichen Ebenen der Parallelisierung vorgestellt. Neben der Parallelisierung der Ausgabe der fertigen Daten, kann sie sowohl auf Systemebene zwischen gleichzeitig laufenden Visualisierungsmethoden als auch innerhalb einer Visualisierungsmethode selbst vorgenommen werden. Für das Letztere gibt es keinen allgemeinen Ansatz, sondern es muss je nach Typ der Visualsierungsmethode ein eigenes Vorgehen für die jeweils optimale Parallelisierung entwickelt werden.

Die Datenverwaltung wird so angelegt, dass sie mehrere Anfragen parallel bearbeiten kann. Zusätzlich wird das progressive Grid Datenformat vorgestellt, das bessere Kompressionsraten und ein positionsabhängiges level of Detail bei der Darstellung verspricht, und damit eines der Kernprobleme der Simulation löst: Die dabei entstehenden großen Datenmengen.

Durch konsequente adaptive Parallelisierung wird die Eigenschaft der Skalierbarkeit der Visualisierung erreicht, die es dem System erlaubt, sich auf verschiedenen Plattformen an unterschiedliche Hardwareausstattungen anpassen zu können.

Das vorgestellte Konzept erlaubt neben der Optimierung durch Parallelisierung auch die Integration von neuen Visualisierungsmethoden. Durch eine allgemeine Schnittstelle und eine entsprechende Funktionskapselung im Zusammenspiel mit einer dazugehörigen Factory, ist es jederzeit möglich, auf einfache Art und Weise das Visualisierungssystem um neue Visualisierungsmethoden zu erweitern. Das Visualisierungssystem kann somit schnell mit zusätzlichen Darstellungsmöglicheiten für Simulationsdaten oder speziellen Ausprägungen bekannter Methoden ergänzt werden.

# Kapitel 6

# Spezifikation und Realisierung

Das folgende Kapitel beschreibt die Umsetzung der in Kapitel 5 vorgestellten Konzepte. Für die wichtigsten Komponenten des Visualisierungssystems für Strömungsdaten wird ihre Realisierung spezifiziert. Am Ende des Kapitels wird noch einmal konrekt auf die Realisierung einiger spezieller Visualisierungsmethoden eingegangen, die auch im Rahmen dieser Arbeit entwickelt und umgesetzt wurden.

## 6.1 Verwendete Werkzeuge

Bei der Auswahl der Realisierungs-Werkzeuge wurde Wert darauf gelegt, dass sie den in Kapitel 4 aufgestellten Anforderungen nicht durch spezielle Auflagen im Wege stehen. Insbesondere die Plattformunabhängigkeit war ein wichtiges Entscheidungskriterium bei ihrer Selektion.

Neben der Programmiersprache C++ [Str86, Str91, SE90] wurden für die Realisierung die bereits im Grundlagen-Kapitel 3 vorgestellten Werkzeuge wie z.B. OpenGL [SGIc] und Coin [COI] bzw. OpenSG [OPEc] zur Realisierung verwendet.

Für die Realisierung der graphischen Oberfläche wurde die Hilfsbibliothek QT [QT] herangezogen. QT stellt im Kern ein Framework für die Gestaltung und Implementierung von Benutzeroberflächen zur Verfügung. Dabei wurde QT kosequent so ausgelegt, dass die einzelnen Komponenten Plattform unabhängig sind. Es ist für alle wichtigen Plattformen (Windows, Linux, Unix, MacOS) verfügbar. Ferner existieren für viele Szenegraph Bibliotheken direkte Anbindungen bzw. Fensterklassen, die deren Integration in eine QT GUI-Umgebung vereinfachen. Die wichtigste Klasse in QT ist das sogenannte *QObject*. Sie enhält alle wesentlichen Bestandteile, die eine Zusammenarbeit mit QT erlauben. QT bietet ein integriertes Nachrichten / Event System, mit dem sich Nachrichten zwischen einzelnen QObjects und zwischen einzelnen Threads transparent versenden lassen. Für die notwendige Ereignissteuerung der GUI bietet QT entsprechende Mechanismen an. Verschiedene Programmteile können durch das Schicken von Ereignissen (Events) miteinander kommunizieren oder man kann an passender Stelle ein *Signal* auslösen, so dass eine vorher mit dem Signal verbundene Methode,

der sogenannte *Slot*, aufgerufen wird. Beide Varianten können gemischt benutzt werden und sind flexibel anpassbar.

Ein weiterer wichtiger Punkt für die Realisierung ist die Threadabstraktion, da sie neben dem Nachrichtenversand eine der zentralen Grundlangen des Systems bildet. Auch hier gibt es zahlreiche Plattform unabhängige Varianten, wie sie z.B. in den Omnithreads (Bestandteil der OmniORB Library [OMN]), in ACE [Sch], QT [QT], oder auch in OpenSG [OPEc] implementiert sind. Bei der Realisierung des Visualisierungssystems wurden verschiedene dieser Threadabstraktionen verwendet und diese jeweils gekapselt. Weitere Details hierzu folgen in Unterkapitel 6.9.

## 6.2 Design Patterns

Da Design Patterns bei der Beschreibung der Lösung eines Problems von den konkreten Aspekten der Programmierung abstrahieren, ist es erforderlich hierfür eine geeignete Umsetzung zu finden.

Die Realisierung von Design Patterns in C++ und unterschiedliche Varianten in der Implementierung ist in [Ale01] ausführlich beschrieben. Die Realisierung im Zusammenhang mit dem in dieser Arbeit entwickelten Visualisierungssystems für Strömungsdaten steht in enger Anlehnung daran. In den folgenden drei Unterkapiteln wird die Umsetzung der drei hierbei verwendeten Design Patterns kurz genauer beschrieben.

### 6.2.1 Singleton

In [Ale01] werden viele Details zur Implementierung von Singletons sehr ausführlich dargestellt. Im folgenden wird deshalb nur auf die zwei für diese Arbeit wesentlichen Aspekte eingegangen:

- Garantie maximal einer Instanz
- Sichere Verwendung von Singletons mit mehreren Threads

#### 6.2.1.1 Garantie maximal einer Instanz

Um sicherzustellen, dass kein Benutzer einer Klasse eine eigene Instanz der Klasse erzeugen kann, muss der Konstruktor der Klasse *private* sein. Außerdem muss die automatische Erzeugung eines Copy-Konstruktors und Copy-Operators verhindert werden, damit die existierende Instanz nicht kopiert werden kann.

Um dennoch einer anderen Klasse den Zugriff auf das Singleton Objekt zu ermöglichen, muss in der Singleton Klasse eine statische Funktion existieren, über die die Instanz erreicht werden kann. Da statische Funktionen einer Klasse auch ohne instanziertes Objekt immer

aufrufbar sind, auch wenn noch keine eigenständige Instanz des Objektes angelegt wurde. Existiert noch keine solche Instanz, ist diese statische Funktion dafür verantwortlich, die Instanz zu erzeugen. Existiert die Instanz bereits, dann liefert die Funktion einfach eine Referenz auf diese Instanz zurück.

**Verwendetes Basiskonstrukt zur Realisierung von Singletons in C++**

```
class Singleton
{
    public:
        static Singleton& instance()
        {
            if(!pInstance)
                pInstance = new Singleton();
            return *pInstance;
        }

    private:
        Singleton(void);
        Singleton(const Singleton&);
        Singleton& operator = (const Singleton&);

        static Singleton* pInstance;
}
```

Wie beschrieben wird in der statischen Funktion *instance()* geprüft, ob bereits ein Objekt der Klasse existiert. Falls ja, wird die entsprechende Referenz auf dieses Objekt zurückgeliefert. Falls nein, dieses Objekt neu angelegt. Der Konstruktor der Klasse ist als private deklariert und damit von außen nicht zugreifbar.

Durch diese zwei Mechanismen (privater Konstruktor und statische Funktion) wird sichergestellt, dass nie mehr als eine Instanz der Klasse existiert, aber die einzig existierende Instanz für alle erreichbar ist.

#### 6.2.1.2 Singleton und Multithreading

Gerade bei der Verwendung von Singletons mit mehreren Threads muss darauf geachtet werden, dass Daten nicht durch Race-Conditions (vgl. Kapitel 3.4.2.1) verloren gehen oder zerstört werden.

Ein Problem hierbei, ist die Erzeugung der Singleton Instanz. Es muss über entsprechende Synchronisations-Mechanismen (z.B. Semaphore) sichergestellt werden, dass nicht versehentlich mehrere Instanzen desselben Singletons erzeugt werden, da der Aufruf der statischen *instance()* Funktion bespielsweise aus mehreren Threads gleichzeitig erfolgt. Falls noch kein

entsprechendes Objekt exisitert, würden alle diese Threads gleichzeitig feststellen, dass eines instanziiert werden muss. Das würde zu entsprechenden Inkonsistenzen führen. An dieser Stelle muss also das neue Erstellen des Objektes vor gleichzeitigem Doppelaufruf geschützt werden. In [Ale01] wird hierfür das *Double Checked Locking Pattern* vorgestellt, das die Erzeugung nur einer Instanz bei Verwendung mehrerer Threads garantiert, ohne zusätzlichen Overhead, wenn die Instanz bereits existiert.

Beim *Double Checked Locking Pattern* wird als erstes überprüft, ob die Singleton Instanz bereits existiert. Wenn dies nicht der Fall ist, wird mit Hilfe eines statischen Mutex Objektes sichergestellt, dass nicht mehrere Threads die nachfolgenden Operationen gleichzeitig ausführen können. Anschließend muss nochmals überprüft werden, ob bereits eine Instanz existiert (es könnte ja nach der ersten Überprüfung und vor dem Betreten des gegenseitig ausgeschlossenen Codes ein anderer Thread die Instanz erzeugt haben). Falls sie immer noch nicht existiert, wird die Instanz erzeugt.

**Das *Double Checked Locking Pattern* in C++**

```
Singleton& Singleton::instance()
{
    if(!pInstance)
    {
        mutex.lock();
        if(!pInstance)
            pInstance = new Singleton();
        mutex.unlock();
    }
    return *pInstance;
}
```

Damit bei Verwendung des Singletons aus mehreren Threads keine Daten zerstört werden können, müssen auch die kritischen Teile der Methoden des Singletons unter gegenseitigem Ausschluss arbeiten. Das kann ebenfalls durch die Verwendung geeigneter Synchronisations-Mechanismen erreicht werden.

### 6.2.2 Factory

Bei der Realisierung des Factory Patterns wird für jede Gruppe von Klassen (Basisklasse und von ihr abgeleitete Klassen) eine eigene Factory benötigt. Diese Factory kann für sich alleine keine Objekte erzeugen, sie wird erst im Zusammenspiel mit den entsprechenden Erzeugern wirkungsvoll.

Für jede Klasse, die mit Hilfe der Factory instanziiert werden soll, muss bei der Factory ein Erzeuger registriert werden. Dieser Erzeuger hat die Aufgabe, ein Objekt der zugehörigen

Klasse anzulegen. Normalerweise wird jeder Erzeuger zusammen mit einer ID (z.B. ein ganzzahliger Wert oder ein String) bei der Factory registriert, mit dessen Hilfe das zu erzeugende Objekt spezifiziert wird. Eine solche ID kann aber auch automatisch von der Factory für alle Erzeuger vergeben werden, was eine bequemere Benutzung dieses Mechanismus verspricht: Das Wissen über zur Verfügung stehende IDs muss nicht auf verschiedene Klassen verteilt implementiert werden.

### 6.2.3 Command

Bei der Realisation des Command Patterns helfen sogenannte *Funktoren* [Ale01]. Ein Funktor ist ein Objekt, das es erlaubt, eine „auszuführende Operation“ typsicher in einem Objekt zu speichern. Eine „auszuführende Operation“ kann hierbei unterschiedliche Dinge bedeuten. Zu nennen sind beispielsweise

- Aufruf einer einfachen Funktion (wie in C)
- Aufruf einer Methode eines Objektes
- Aufruf eines anderen Funktors

In [Ale01] werden *generalisierte Funktoren* vorgestellt, mit deren Hilfe alle der oben genannten Operationen ausgeführt werden können. Diese bieten auch die Möglichkeit, mehrere Funktoren in Reihe auszuführen (sogenanntes *Chaining*) und Parameter von Funktionsaufrufen mit festen Werten zu verknüpfen (*Binding*).

## 6.3 Proben

Die Proben stellen die direkte Umsetzung des gleichnamigen Konzeptes, das in Kapitel 5.2.1 vorgestellt wurde, dar. Sie enthalten die im Konzept dargelegten Eigenschaften als Attribute und zusätzlich noch entsprechende Funktionen, um auf diese zugreifen zu können. Im Folgenden werden die wichtigsten Funkionen der Klasse kurz vorgestellt. Sofern es dem Verständnis dient, wird auf einige dieser Funktionen kurz eingegangen. Die im Rahmen dieser Realisierung implementierte Probenklasse besteht aus einem rechteckigen Datenraumausschnitt.

---

**Die *Probe* Klasse**

---

```
class Probe : public QObject
{
    Q_OBJECT

    public:
        Probe(DataSource* dS, VisModule* vM, CoordSystem* cS);
```

```
        ~Probe();

        DataSource* getDataSource();
        VisModule* getVisModule();
        CoordSystem* getCoordSystem();

        void update(void);
        NodeType* getNode();

        bool getDisabled();
        void setDisabled(bool disable);

        void resize(Vector3f* size);
        void rotate(PVQuaternion* rotation);
        void translate(Vector3f* vec);

        void setFrame (unsigned int frame);

        double getTransparency();
        void setTransparency(double transparency);

        //...

};
```

---

Die Klasse selbst wird von QObjekt abgeleitet. Damit erbt sie automatisch die entsprechenden typischen QObjekt Eigenschaften, d.h. die Möglichkeit QT Signals und Slots bereitzustellen.

`Probe(DataSource* dS, VisModule* vM, CoordSystem* cS);`
Mit diesem Konstrutor wird die Probe angelegt. Sie bekommt ein eigenes Koordinatensystem zugeordnet. Zusätzlich wird eine Datenquelle und eine Visualisierungsmethode mit ihr verbunden (Abbildung 6.1).

`void update(void);`
Über diese Funktion wird die Probe dazu veranlaßt, die angeschlossene Visualisierungsmethode bzw. deren Berechnungen neu anzuwerfen. Die Visualisierungsmethode berechnet ihre Daten daraufhin neu aus der Datenquelle in Abhängikeit ihrer Position und dem aktuellen Zeitschritt.

`NodeType* getNode();`
Durch Aufruf dieser Routine, erhält das System einen neuen Szenegraphknoten der mit der Probe verbundenen Visualisierungsmethode. Hierbei ist der Knotentyp ein allgemeiner Datentyp, der je nach Rahmenbedingung angepaßt werden kann (z.B. an Coin [COI] oder OpenSG [OPEc]). Der Aufruf wird an die verbundene Visualisierungsmethode durchgereicht.

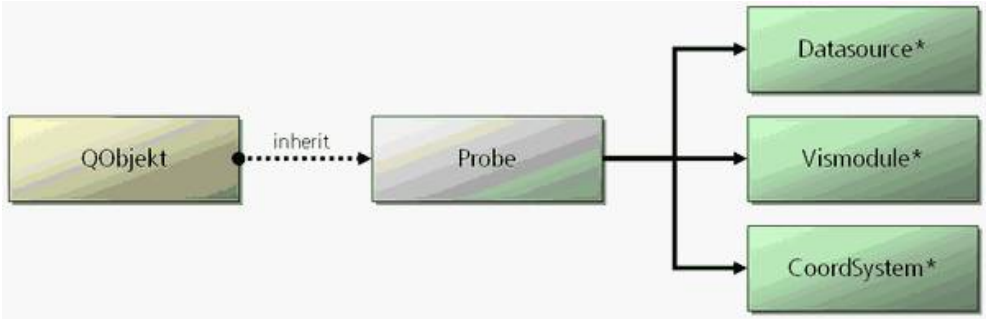



***Abbildung 6.1:*** *Die Probenklasse erbt von QObjekt. Außerdem speichert sie Verweise auf ein lokales Koordinatensystem, eine Datenquelle und eine Visualisierungsmethode.*

Auf Wunsch wird zusätzlich das Koordinatensystem und die Umrandung der Probe (Quader) dem entstandenen Knoten hinzugefügt.

```
void setDisabled(bool disable);
```
Mit dieser Funktion ist es möglich, die Probe zur Laufzeit zu „deaktivieren“, ohne sie zu löschen. Dadurch kann man beispielsweise gezielt einzelne Proben aus der Szene ausblenden. Die mit ihrer Aktualisierung verbundene Rechenzeit wird frei. Jedoch werden die Daten der Probe im Speicher gehalten. So kann durch Aktivierung ihre Darstellung sehr schnell wieder verfügbar gemacht werden, da nicht alle zugehörigen Kalkulationen / Objekterzeugungen neu durchgeführt und ihre Parameter neu justiert werden müssen.

```
void resize(Vector3f* size);
void rotate(PVQuaternion* rotation);
void translate(Vector3f* vec);
```
Mit Hilfe dieser drei Funktionen läßt sich die Probe im Datenraum ausrichten und orientieren. Sie kann in ihrer Größe verändert, verschoben oder rotiert werden. Das hat im Anschluß direkten Einfluß auf das Daten Clipping. Die Darstellung der Visualisierungsmethode wird auf den Bereich beschränkt.

```
void setFrame (unsigned int frame);
```
Durch diese Funktion hat das System die Möglichkeit, die Darstellung in einem zeitlich veränderlichen Datensatz auf einen bestimmten Zeitschritt zu setzen. Diese Funktion ist für jede Probe einzeln verfügbar. So wird es beispielsweise möglich, dass verschiedene Proben nicht nur räumlich unterschiedliche Bereiche des Datensatzes visualisieren, sondern auch welche, die sich bezüglich der Simulationszeit unterscheiden.

```
double getTransparency();
void setTransparency(double transparency);
```
Diese Funktion übermittelt den aktuellen Transparenzwert der Darstellung an die Probe. Der Wert wird zum einen auf die angezeigte Umrandung / das dargestellte Koordinatensystem der Probe selbst angewendet, als auch an die angeschlossene Visualisierungsmethode weitergegeben.

### 6.3.1 Probepool

Der Probepool ist die Umsetzung des Visualization Method Pool Konzeptes. Er speichert die zur Laufzeit aktiven Proben in einer Containertypischen Form.

Zunächst wird eine spezielle allgemeine *Pool-Containerklasse* implementiert. Diese bekommt die grundsätzlichen Funktionen, die zur Verwaltung einer unsortierten Menge von instanziierten Objekten gleichen Typs benötigt werden. Hierzu zählen z.B. die folgenden Funktionen:

**add**
: Mit Hilfe dieser Methode können neue Objekte in den Pool zur Speicherung gegeben werden.

**remove**
: Diese Funktion dient der Löschung eines speziellen Objektes aus dem Pool.

**iterator**
: Durch Verwendung von Iteratoren ist es möglich, von einem Objekt des Pools zum nächsten zu springen. Die Objekte werden so abgelegt, dass dabei kein Objekt zweimal besucht wird.

**iterator::begin**
: Mit diesem speziellen Iterator kann man auf das erste Element im Pool springen.

**iterator::end**
: Mit diesem speziellen Iterator kann man auf das letzte Element im Pool springen.

Durch Erben von der Pool-Containerklasse wird die Probepool Klasse spezifiziert (Abbildung 6.2). Sie wird als Singleton angelegt, da es wichtig ist, nur einen (gültigen) Probepool im System zu haben. Entsprechend muss verhindert werden, dass mehrere solcher Pools gleichzeitig angelegt werden können. Es werden Funktionen zum Einspeichern von neuen Proben, zu deren Abfrage und Löschen integriert. Ansonsten wird diese Klasse übersichtlich gehalten und nicht mit Zusatzfunktionalität überladen. Über den Probepool sind die zur Laufzeit aktiven Proben für den Rest des Systems zugänglich.

## 6.4 Generische Visualisierungsmethode

Die Visualisierungsmethoden werden wie konzipiert von einer generischen Mutterklasse abgeleitet, bevor sie in das System integriert werden. Auf alle Funktionen dieser allgemeinen Klasse an dieser Stelle einzugehen würde den Rahmen dieses Textes sprengen. Aus diesem Grund werden an dieser Stelle jetzt nur die wichtigsten von ihnen erläutert.

**Die *Vismodule* Klasse**

```
class VisModule
{
    public:
        VisModule(Probe* probe);
        virtual ~VisModule();

        void setNumberOfThreads(unsigned int threadNumber);

        void addClipPlane();
        void clearClipPlanes();
        int getNumberOfClipPlanes();
        void enableCutPlane(bool enable);

        virtual void update();
        virtual NodeType getNode();

        virtual void setFrame(unsigned int);

        double getTransparency();
        void setTransparency(double transparency);

        virtual const VisModuleType* getType() const = 0;

        //...

};
```

Die Klasse, die eine allgemeine Implementierung der Visualisierungsmethoden im System bereitstellt, wird als *VisModule Klasse* bezeichnet.

`VisModule(Probe* probe);`
Hiermit wird die Visualisierungsmethode bzw. eine Objektinstanz zur Programmlaufzeit angelegt. Alle wichtigen Parameter werden initialisiert und mit sinnvollen Werten, die sich je nach Visualisierungsmethode unterscheiden, gefüllt. Eine Referenz auf die zugehörige Probe wird mit übergeben.

`void setNumberOfThreads(unsigned int threadNumber);`
Hier wird die Anzahl der zu verwendenen Threads für die Berechnungen der Visualisierungsmethode festgelegt. Die Anzahl zu verwendender Threads wird absichtlich nicht systemweit mit einem festen Wert vorgegeben. Auf diese Weise läßt sich deren Anzahl invididuell pro Visualisierungsmethode steuern, wird also zu einem weiteren Steuerparameter gemacht. So hat man den Vorteil, einzelne Visualisierungsmethoden zur Laufzeit z.B. durch Vergabe von

mehr Threads gegenüber anderen Methoden zu „bevorteilen“ oder gezielt auf auftretende Leistungsschwankungen zu reagieren.

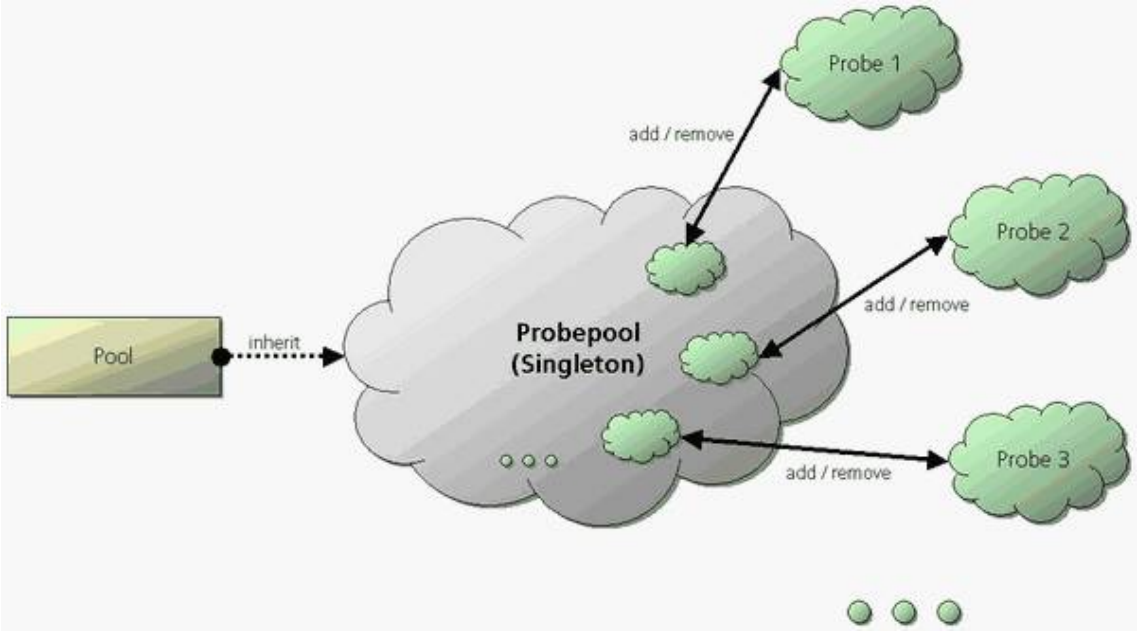



***Abbildung 6.2:*** *Der Probepool erbt von der Pool Containerklasse. In ihm werden die zur Laufzeit aktiven Proben gespeichert.*

```
void addClipPlane();
void clearClipPlanes();
int getNumberOfClipPlanes();
void enableCutPlane(bool enable);
```

Mit diesen Funktionen hat der Benutzer die Möglichkeit, *Clipplanes* auf die Visualisierungsmethode anzuwenden. Clipplanes sind Hyperebenen im Datenraum bzw. in der Probe, die die räumliche Darstellung, d.h. die Darstellungsgeometrie der Visualisierungsmethode an einer vorgegebenen Grenze „abschneiden“. Zusätzlich zu den Clipplanes werden noch *Cutplanes* realisiert. Diese dienen nicht dazu, die Darstellung an einer bestimmten Hyperebene „abzuschneiden“, sondern sie schneiden eine „dünne Scheibe“ aus der Darstellung heraus.

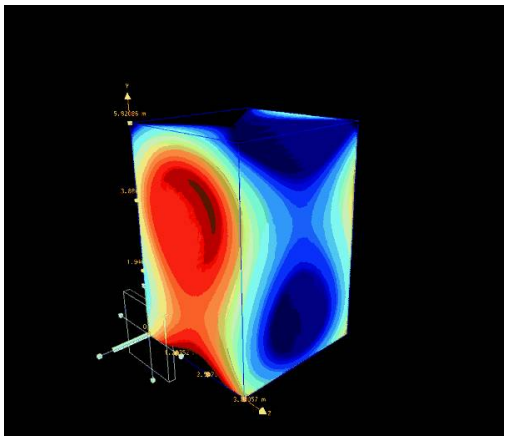
**Definition 6.1 (Clipplane)** *Mit Hilfe von Clipplanes kann man die virtuelle Szene an einer vorgebenen Hyperebene „abschneiden“. Alles jenseits dieser Hyperebene wird nicht dargestellt.*

**Definition 6.2 (Cutplane)** *Eine Cutplane erlaubt es, aus der dargestellten Szene eine „dünne Scheibe“ herauszuschneiden. Nur diese wird visualisiert.*

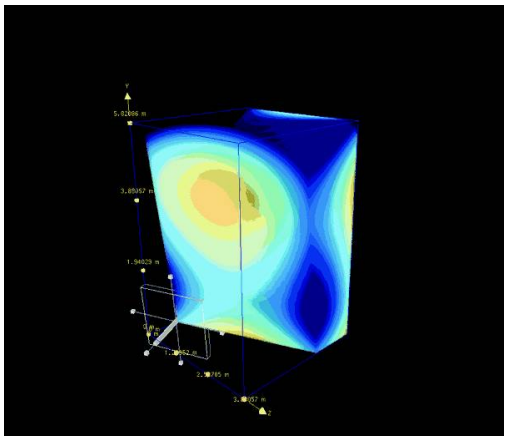
Eine Szene, die einen Beispieldatensatz enthält, dessen aktive Visualisierungsmethode, d.h. Probe, mit Clip- und Cutplanes beschnitten wird, ist in Abbildung 6.3 zu sehen.

```
virtual void update();
```

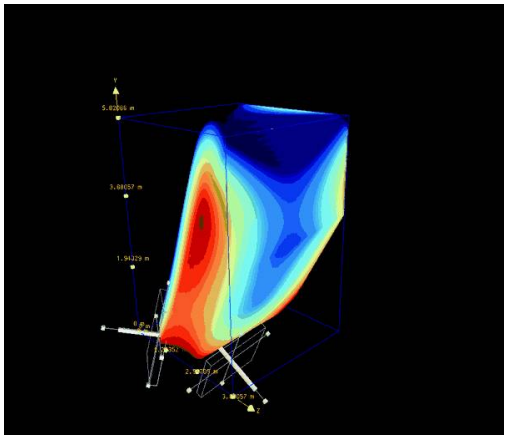
Diese Funktion wird von der update Funktion der verbundenen Probe aufgerufen, wenn



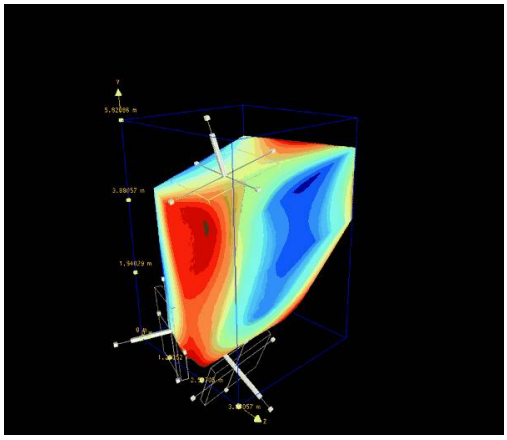
1: Die komplette Visualisierung ohne Clip- & Cutplanes



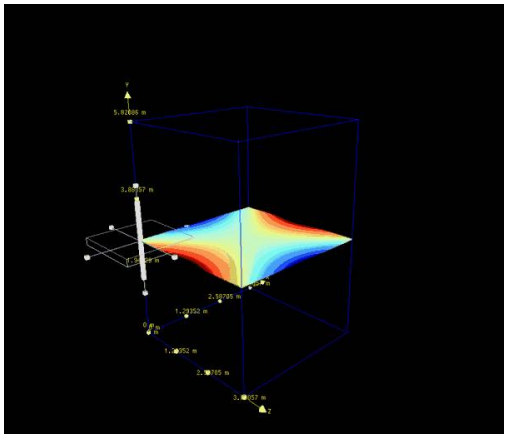
2: Eine Clipplane wird durch den Datensatz bewegt



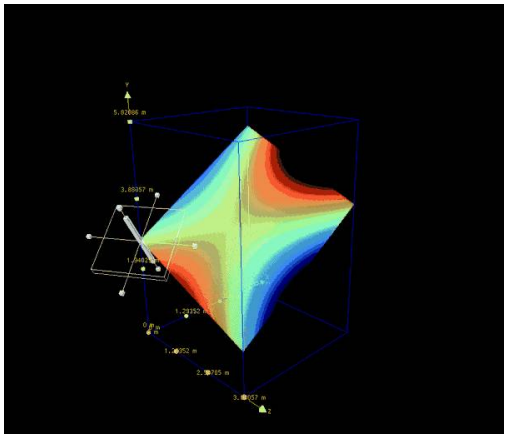
3: Eine zweite Clipplane kommt hinzu



4: Drei Clipplanes sind gleichzeitig aktiv



5: Eine Cutplane schneidet eine dünne Scheibe aus der Visualisierung



6: Dieselbe Cutplane, nun leicht gedreht

***Abbildung 6.3:*** *Clip- & Cutplanes: Die Volumen-Rendering Visualisierung innerhalb der quaderförmig umrandeten Probe wird auf verschiedene Art und Weise beschnitten.*

eine Neuberechnung der Visualisierungsdaten erforderlich wird. Die Berechnungen laufen in der vorgegebenen Anzahl von Threads ab.

`virtual NodeType getNode();`
Hier wird der Szenegraphknoten der Visualisierungsmethode zugänglich gemacht. Die verbundene Probe kann ihn hierüber abfragen und dem System zur Verfügung stellen. Im späteren Programmverlauf wird sich der Central Scheduler (Kapitel 6.6) darum kümmern, dass dieser Knoten an der korrekten Stelle im Ausgabeszenegraph platziert wird.

`virtual void setFrame(unsigned int);`
In dieser Funktion wird, wie dies auch schon in der übergeordneten Probe der Fall ist, der aktuell darzustellende Zeitschritt des Datensatzes an die Visualisierungsmethode übergeben.

```
double getTransparency();
void setTransparency(double transparency);
```
Genau wie schon in der übergeordneten Probe wird hier die Transparenz der Visualisierungsmethode geregelt. Zulässige Werte sind zwischen 0.0 und 1.0 und regeln direkt die Durchsichtigkeit / Opazität Wert der Darstellung.

`virtual const VisModuleType* getType() const = 0;`
Über diese Funktion wird der Typ der Visualisierungsmethode abfragbar gemacht. Dazu mehr im nächsten Unterkapitel.

### 6.4.1 Typklasse

Die Typklasse *VisModuleType*, die über die entsprechende Funktion innerhalb einer Visualisierungsmethode abrufbar ist, stellt Informationen der sich dahinter verbergenden Visualisierungsmethode für das System bereit. Um das Wissen über vorhandene Visualisierungsmethoden-Typen verteilt pflegen zu können, wird hier absichtlich statt einer einfachen ID (z.B. per *enum*) ein Objekt zur Typrepräsentation verwendet. Durch Vererbung dieser Klasse ist es auf einfache Art und Weise möglich neue Typen einzuführen. Die Typklasse speichert neben einer eindeutigen ID auch noch andere Informationen, die hier kurz aufgelistet werden:

**ID**
: Eine eindeutige ID pro Visualisierungsmethode. Wenn eine neue Methode dem System hinzugefügt wird, wird hier eine neue ID eingetragen.

**Name**
: Der Name, der im System für die Visualisierungsmethode verwendet werden soll (z.B. „Marching Cubes basierte Iso-Fläche“ oder „3D Textur Volumenrendering“). Dieser Name wird dem Benutzer angezeigt, wenn er eine Liste der zur Verfügung stehenden Visualisierungsmethoden vom System präsentiert bekommt.

**Benötigte Extenions**
: Einige in das System integrierte Visualisierungsmethoden sind auf bestimmte Graphikkarten Extensions optimiert. In diesem Teil des Visualisierungemethoden-Typs wird

gespeichert, welche dieser Extensions von der jeweiligen Visualisierungsmethode vorausgesetzt werden. Sollten diese nicht auf dem gegenwärtigen System verfügbar sein, so wird die entsprechende Visualisierungsmethode nicht zugänglich gemacht.

**Kompatibilitätstest**

Diese Funktion hat als Eingabe Parameter eine Referenz auf eine Datenquelle. Hier wird getestet, ob die entsprechende Visualisierungsmethode mit der übergebenen Datenquelle „kompatibel“ ist, d.h. ob sie imstande ist sie zu interpretieren und damit anzuzeigen. Visualisierungsmethoden, die ein Vektorfeld als Grundlage brauchen, sind z.B. nicht fähig, ein Skalarfeld zu visualisieren.

### 6.4.2 Vismodule Factory

Die Vismodule Factory ist die Instanz im Visualisierungssystem, bei dem sich alle verfügbaren Visualisierungsmethoden registrieren müssen. Hierfür stellt sie eine Registrierfunktion bereit, über die die entsprechende Methode sich selbst anmelden kann. Auch das Abmelden wird unterstützt. Wie ihr Name schon sagt, wird sie als Factory angelegt. Sie enthält einen Container, der die angemeldeten Methoden speichert. Um zu verhindern, dass zur Laufzeit mehrere Factories erzeugt werden, wird auch diese Klasse als Singleton implementiert (Abbildung 6.4).

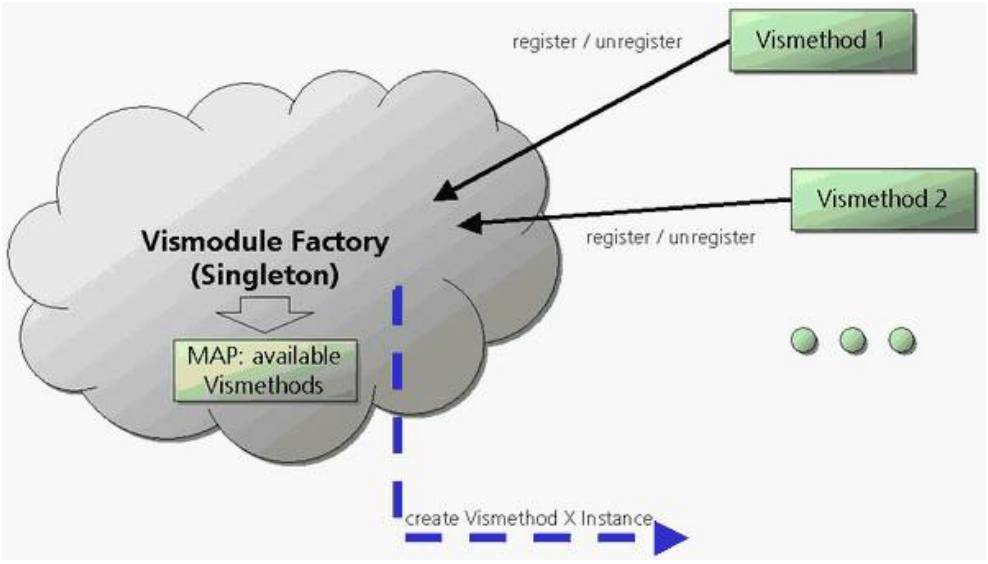



***Abbildung 6.4:*** *Vismodule Factory: Bei ihr werden die zur Verfügung stehenden Visualisierungsmethoden über deren Erzeugerfunktion registriert und in einem Containerspeicher (MAP) abgelegt. Soll zur Laufzeit ein Objekt einer Visualisierungsmethode instanziiert werden, so wird über diese Factory die entsprechende Erzeugerfunktion aufgerufen.*

## 6.5 Actions

Actions realisieren die im Konzept (Kapitel 5.3.1.1) vorgestellte Event Implementierung, die sowohl den auszuführenden Befehl, als auch zugehörige Parameter transportieren können. Sie werden zur Steuerung des Systems verwendet (z.B. indem einzelne Parameteränderungen an ausgewählte Proben / Visualisierungsmethoden übermittelt werden).

Zu ihrer Realisierung wird das Command Pattern verwendet, um Aktionen speichern und später ausführen zu können. Die zu speichernden Aktionen haben dabei die Besonderheit, dass sie Methoden eines Objektes sind, die genau einen fixen Parameter brauchen. Dieser Parameter ist bei der Erzeugung der Action bekannt und wird zusammen mit der Action gespeichert. Hierdurch lassen sich Aktionen speichern und später ohne weitere Parameter ausführen.

**Die *Action* Klasse**

```
class Action
{
    public:
        Action()
        virtual ~Action()

        virtual void operator () () = 0;
};
```

Die Action Klasse ist im Endeffekt eine „leere“ Hülle, die nur eine wichtige Funktion bereitstellt: Die *operator* genannte Funktion.

Damit Aktionen möglichst einfach zu erzeugen und zu verwenden sind, wird ein Template entwickelt, mit dessen Hilfe beliebige Methoden (hier mit nur einem Parameter) von beliebigen Objekten einfach verwendet werden können. Zur internen Speicherung der Action werden dann Funktoren verwendet.

**Die *Funktor* Klasse**

```
template<typename ObjectPointerType, typename MemberFunctionType
    , class ParamType>
class MemberFunctor
{
    public:
        MemberFunctor(ObjectPointerType objectPointer,
            MemberFunctionType memberFunction)
        : mObjectPointer(objectPointer), mMemberFunction(memberFunction)
```

```
    void operator () (ParamType param)

        (mObjectPointer->*mMemberFunction)(param);


    private:
        ObjectPointerType mObjectPointer;
        MemberFunctionType mMemberFunction;
};
```

Der Funktor besitzt zwei wichtige Attribute: Das zugehörige Objekt und die auf ihm anzuwendente Funktion. Bei Aufruf der Funktion wird ein Parameter mit übergeben.

Hier dann die entsprechende Action Template Klasse:

**Die *ObjectAction* Klasse**

```
template<typename ObjectType, typename MemberFunctionType, class ParamType>
class ObjectAction : public Action
{
    public:
        typedef ObjectType* ObjectPointerType;
        typedef MemberFunctor<ObjectPointerType,MemberFunctionType,
            ParamType> Functor;

        ObjectAction(MemberFunctionType Function,
            ObjectPointerType objectPointer, ParamType param)
        : mFunctor(objectPointer, Function), mParam(param)



        void operator () ()

        mFunctor(mParam);


    private:
        Functor mFunctor;
        ParamType mParam;
};
```

Die *ObjectAction* speichert den zu übertragenden Parameter als auch die zugehörige Funktion im angegebenen Objekt. Somit ist die generische Actionklasse implementiert. Eine exemplarische Action, die nun von der allgemeinen Action Klasse erbt, sieht dann folgendermaßen aus:

**Die *ProbeResizeAction* Klasse**

```
class ProbeResizeAction : public ObjectAction<Probe,void(Probe::*)
    (const Vector3f&),Vector3f>
{
    public:
        typedef ObjectAction<Probe,void(Probe::*)(const Vector3f&),
            Vector3f> ProbeAction;

        ProbeResizeAction(Probe* probe, const Vector3f& resize)
        : ProbeAction(&Probe::resize, probe, resize)
};
```

Beim Auswerten diese Action wird die Funktion `Probe:resize()` aufgerufen und bekommt einen Parameter vom Typ `Vector3f` übermittelt.

Um diese Action nun im System abzubilden, bedient man sich der `postEvent()` Funktion in QT. Sie ermöglicht es, innerhalb eines Prozesses Events an von QObjekt geerbte Objekte zu verschicken. Dabei ist der erste Parameter ein QT Event - d.h. eine Nachricht. Da man mit Hilfe der `posttt()` Funktion allerdings nur `QEvents` (die Event Klasse von QT) verschicken kann, wird es nötig, noch eine weitere Klasse anzulegen, die von der QEvent Klasse bzw. der `QCustomEvent` Klasse erbt:

**Die *ProbeActionEvent* Klasse**

```
class ProbeActionEvent : public QCustomEvent
{
    public:
        ProbeActionEvent(Probe* probe, Action* action=NULL)
        : QCustomEvent(2000), mProbe(probe), mAction(action)



        Probe* getProbe()  return mProbe;
        Action* getAction()  return mAction;

    private:
        Probe* mProbe;
        Action* mAction;
};
```

Diese „ProbeActionEvent“ genannte Klasse kapselt sowohl die weiter oben definierte Action, also auch die Probe, an die die Action gesendet werden soll.

Eine Beispielzeile, die nun diese Action im Programm verschickt, sieht dann also so aus:

```
Action* action = new ProbeResizeAction(probe, size);
QApplication::postEvent(&UpdateManager::instance(),
new ProbeActionEvent(probe, action));
```

Der ganze Zusammenhang der einzelnen Klassen ist auch noch einmal in Abbildung 6.5 zu sehen. Die Vorstellung der `UpdateManager` Klasse folgt in den anschließenden Kapiteln.

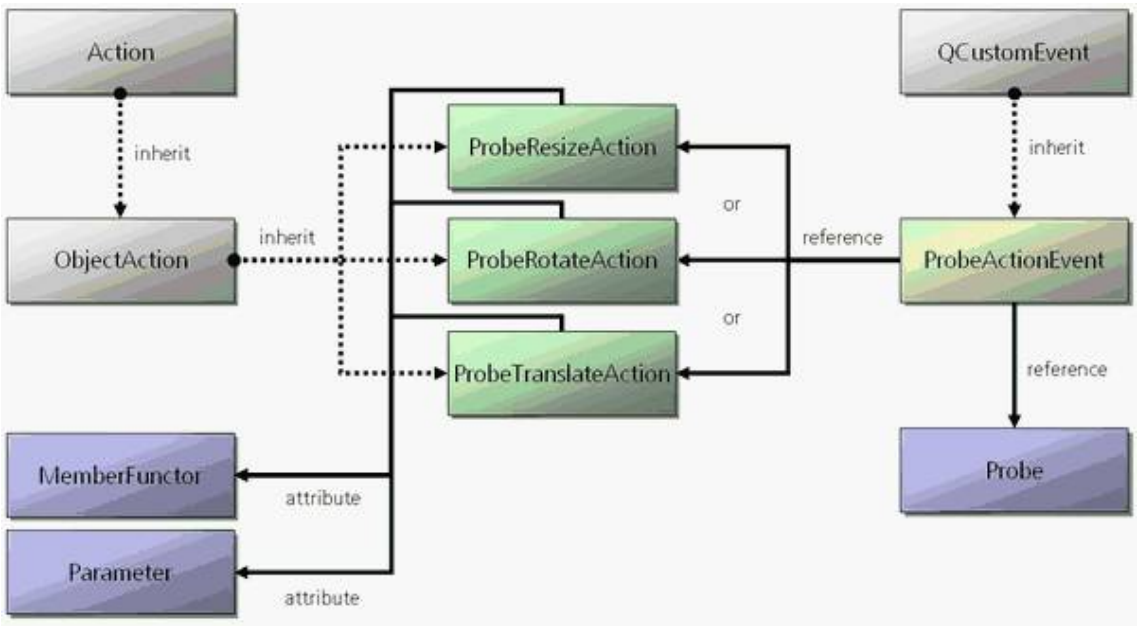



***Abbildung 6.5:*** *Implementierung der `Action` Klasse.*

Neben den allgemeine ProbeActions, die zur generellen Steuerung der Proben dienen (z.B. Translation oder Größenänderung), werden nun zusätzlich noch Actions eingeführt, die speziell auf einzelne Visualisierungsmethoden abgestimmt sind. Durch sie hat das System die Möglichkeit, einzelne Parameter einer Visualisierungsmethode zu steuern.

Als Beispiel sei hier eine Action aufgeführt, mit deren Hilfe man die Auflösung in X Richtung der Line Integral Convolution Visualisierungsmethode steuern kann:

---

**Die *LICResolutionXAction* Klasse**

---

```
class LICResolutionXAction : public ObjectAction<LineIntegralConvolution,
    void(LineIntegralConvolution::*)(unsigned int),unsigned int>
{
    public:
        typedef ObjectAction<LineIntegralConvolution,
            void(LineIntegralConvolution::*)(unsigned int),
            unsigned int> LICAction;
```

```
        LICResolutionXAction(LineIntegralConvolution* lic, unsigned int res)
        : LICAction(&LineIntegralConvolution::setResolutionX, lic, res)
};
```

---

Auch diese Klasse erbt von QObjectAction. Mit Hilfe der ProbeActionEvent Klasse läßt sich nun also auch diese Action im System verschicken.

### 6.5.1 Events

Neben den Actions, die dafür ausgelegt sind sowohl eine Referenz auf eine zugehörige Probe / Visualisierungsmethode, als auch ein entsprechendes Kommando mitsamt Aufrufparameter zu transportieren, ist es an vielen Stellen nötig, lediglich systemweite Nachrichten zu verschicken, ohne eine entsprechende Probe / Visualisierungsmethode damit zu verbinden. Hier genügt es in der Regel, neben der eigentlichen Information, welche Aufgabe zu erledigen ist, einfach eine feste Anzahl weiterer Paramater mit zu übergeben. Für solche Nachrichten wird einfach die `QCustomEvent` Funktion beerbt und in die entsprechende abgeleitete Klasse der zu transportierende Parameter gekapselt. Hier eine kurze Übersicht, über die in der Realisierung unterschiedenen Events / Nachrichtentypen:

**CommonEvents**
: Mit CommonEvents werden Nachrichten bezeichnet, die allgemeinen Charakter haben. Hierzu gehören beispielsweise Events, die zum Speichern von Screenshots auf die Festplatte auffordern, oder wenn die gesamte Darstellung per Timer um einen Zeitschritt weiter geschaltet wird, so daß alle aktiven Visualisierungsmethoden auf den neuen Zeitschritt wechseln. In diesem Fall wird von der Nachricht ein Parameter transportiert: der darzustellende Zeitschritt.

**DataSourceEvents**
: DataSourceEvents sind Nachrichten, die entstehen, wenn sich an den Datenquellen im System etwas ändert. So kann beispielsweise eine Datenquelle gelöscht werden, oder ihr Inhalt wird aktualisiert (durch Auslagerungs / Ladevorgänge).

**SpaceMouseEvent**
: SpaceMouseEvents sind spezielle Events, die in das System eingebracht wurden, um die Steuerung der Navigation durch eine 6D Spacemouse[3DC] zu ermöglichen. Hier werden die 6 Raumparameter (Translation in x,y,z und Rotation in x,y,z) transportiert.

**ProbeEvents**
: Die ProbeEvents transportieren Nachrichten, die Proben betreffen. Neben den bereits vorgestellten ProbeActionEvents, die zu einer dedizierten Probe ein Kommando und ein Aufrufparameter transportieren, gehören hierzu auch Nachrichten, die beispielsweise Proben ein-/abschalten, auf die Darstellung eines bestimmten Zeitschrittes einstellen, die Darstellung der Probe an-/ausschalten oder die Darstellung des Koordinatensystems der Probe an-/ausschalten.

**UpdateRunnerEvents**
: UpdateRunnerEvents sind spezielle Events, die den Updatemanager (siehe folgende Kapitel) darüber informieren, dass Berechnungen gestartet werden sollen / beendet wurden. Hierzu später mehr.

## 6.6 UpdateManager

Der in Kapitel 5.3.1 vorgestellte Central Scheduler übernimmt wie konzipiert die Steuerung und Koordination der einzelnen Systembestandteile. Realisiert wird er in der Klasse `UpdateManager`. Da er nur einmal im System existieren darf, wird er ebenfalls als Singleton angelegt. Hier die entsprechende Klasse mit ihren wichtigsten Bestandteilen:

---

**Die *UpdateManager* Klasse**

---

```
class UpdateManager : public QObject
{
    Q_OBJECT

    public:
        static UpdateManager& instance();
        bool event(QEvent* ev);

    signals:
        void sceneChanged();
        void probesChanged(bool);

    private:
        UpdateManager();
        ~UpdateManager();

        void update(Probe* probe, ProbeActionEvent* event=NULL);
        void update(const DataSource* data);

        void runnerFinished(UpdateRunner* runner);

        void remove(Probe* probe);

        // Singleton: prevent copy constructor/assignment
        UpdateManager(const UpdateManager& updateManager);
        UpdateManager& operator = (const UpdateManager& updateManager);

        typedef map<Probe*,UpdateRunner*> RunnerStore;
        RunnerStore mRunners;
```

```
        RunnerStore mSchedule;
        vector<Probe*> mRemoveSchedule;

        //...

};
```

---

Die Singletonklasse erbt von `QObject`. So erhält sie auf einfache Art und Weise die Fähigkeit, Nachrichten / Events zu empfangen. Zusätzlich kann sie QT typische Signals und Slot enthalten. Die wichtigsten Funktionen werden hier kurz erläutert:

`void sceneChanged();`
Dieses Signal wird vom UpdateManager jedesmal ausgelöst, sobald sich die graphische Darstellung der Szene ändert. Damit per „connect“ Befehl verbundene QObjects können daraufhin entsprechend reagieren. Im umgesetzten Beispiel reagiert z.B. ein Unterfenster des Hauptprogrammes, indem es die aufgelisteten Szenebestandteile aktualisiert.

`void probesChanged(bool);`
Sobald sich an der aktiven Probenliste im Probepool etwas ändert, wird dieses Signal ausgelöst. Auch hier können verbundene Klassen reagieren, indem sie z.B. die Liste der aktiven Proben entsprechend anpassen, sobald sie dieses Signal „empfangen“.

```
void update(Probe* probe, ProbeActionEvent* event=NULL);
void update(const DataSource* data);
```
Mit die wichtigsten Funktionen des UpdateManagers sind die „update“ genannten Routinen. Sie werden gestartet, wenn ein ProbeEvent empfangen wird (da sich z.B. ein Parameter der entsprechenden Visualisierungsmethode durch Klick des Users im Fenster verändert hat). Außerdem können sich die zur Verfügung stehenden Datenquellen ändern, beispielsweise durch Ein- oder Auslagerungsvorgänge. Die Update Funktionen haben die Aufgabe, die entsprechenden (Neu-)Berechnungen in den betroffenen Proben bzw. deren Visualisierung zu starten. Hierzu werden dann pro Probe „UpdateRunner“ genannte Threads (siehe Kapitel 6.6.1) angeworfen, die diese Berechnungen koordinieren.

`void runnerFinished(UpdateRunner* runner);`
Sollte eine in einem UpdateRunner gestartete Berechnung beendet sein, so wird diese Funktion aufgerufen. Wie im Konzept geplant, wird daraufin festgestellt, ob sofort eine weitere Berechnung derselben Probe / Visualisierungsmethode gestartet werden muss (durch gepufferte und aufgelaufene Parameteränderungen). Falls ja wird sofort ein neuer UpdateRunner gestartet. Falls nein, wird der entsprechende neue Szenegraphknoten der Probe abgefragt. Der alte Knoten wird durch den neuen ersetzt, d.h. die graphische Darstellung angepaßt.

`typedef map<Probe*,UpdateRunner*> RunnerStore;`
In dieser Map werden die aktiven UpdateRunner zur Laufzeit gehalten. So kann festgestellt werden, ob zu einer bestimmten Probe bereits ein UpdateRunner, d.h. eine Berechnung läuft oder nicht.

### 6.6.1 UpdateRunner

Die Berechnungen, die pro Probe gestartet werden müssen, sobald eine entsprechende Parameteränderung an den UpdateManager übermittelt wird, werden in einer eigenen Threadklasse behandelt. Sie wird angelegt und gestartet, sobald die Update Funktion des UpdateManagers aufgerufen wird.

---

**Die *UpdateRunner* Klasse**

---

```
class UpdateRunner : public WorkThread
{
    public:
        UpdateRunner(Probe* probe);

        virtual void run();

        void addAction(Action* action)  mActions.push_back(action);

        Probe* getProbe()  return mProbe;
        NodeType getOldNode()  return mNode;

    protected:
        Probe* mProbe;
        NodeType mNode;
        vector<Action*> mActions;

    private:
        UpdateRunner();
        UpdateRunner(const UpdateRunner&);
        UpdateRunner& operator=(const UpdateRunner&);
};
```

---

Die UpdateRunner Klasse erbt von einer verallgemeinerten Threadklasse, WorkThread genannt (siehe Kapitel 6.6.2). Sie speichert die zugehörige Probe und den verknüpften Szenegraphknoten (für Erläuterung der NodeType Klasse, siehe Kapitel 6.10). Zusätzlich kann der Thread noch eine Liste zugehöriger Actions speichern. Hier werden die auflaufenden Parameteränderungen gepuffert. Geht eine die Probe betreffende Action ein, während ein UpdateRunner zur entsprechenden Probe bereits läuft, d.h. seine `run()` Methode aufgerufen wurde, wird ein zweiter Thread angelegt, aber noch nicht gestartet. Dessen `mActions` Vektor wird zur Pufferung der eingehenden Aktions benutzt, bis dieser schließlich per `run()` gestartet wird.

### 6.6.2 WorkThread

Die WorkThread Klasse bildet die systemweite Threadabstraktion. Falls systembedingt ein Wechsel der Threadrealisierung nötig wird, müssen nur hier entsprechende Änderungen der Threadimplementierung vorgenommen werden. Beispielsweise setzt der Einsatz des OpenSG [OPEc] Szenegraphen die Verwendung einer eigenen propritäten Threadklasse voraus. Auch der Wechsel von einer Plattform auf eine andere kann, sofern keine QT Threads verwendet werden dürfen, die Anpassung dieser Klasse nötig machen. Durch die Kapselung der im System verwendeten Threads in eine abstrakte Klasse, ist dieser Umstieg schnell und einfach durchzuführen.

Neben der UpdateRunner Klasse, sind die Visualisierungsmethoden die Hauptnutzer der Workthreads. Sobald sie von einem UpdateRunner dazu aufgefordert werden, ihre Daten neu zu berechnen, starten sie eine vorgegebene Anzahl Workthreads für diese Aufgabe.

Das Zusammenspiel zwischen UpdateManager, UpdateRunner und Workthread läßt sich direkt aus Abbildung 5.7 und 5.8 ableiten. Es ist noch einmal in Abbildung 6.6 veranschaulicht.

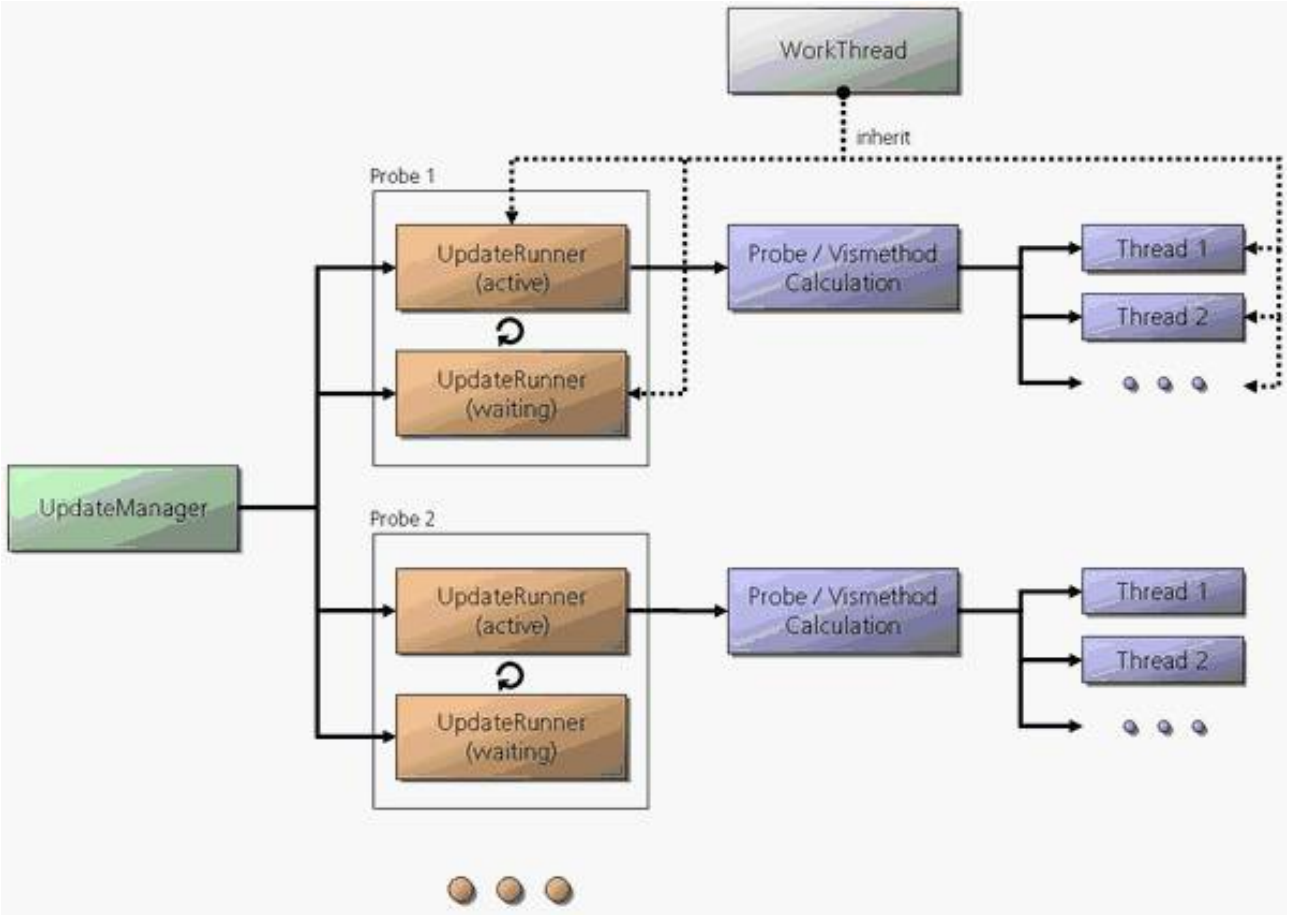



***Abbildung 6.6:*** *Der Zusammenhang zwischen der UpdateManager, UpdateRunner und WorkThread Klasse.*

## 6.7 DataSourceManager

Die Aufgabe der Datenverwaltung im System übernimmt die `DataSourceManager` genannte Klasse. Er speichert alle geladenen und damit für die Visualisierung zur Verfügung stehenden Datenquellen in einer `Storage` genannten Map. Er wird als Singleton angelegt, um die mehrfache Instanziierung zur Laufzeit zu verhindern. Die Daten selbst werden in der Klasse `DataSource` gehalten, die im Anschluß vorgestellt wird.

**Die *DataSourceManager* Klasse**

```
class DataSourceManager
{
    public:
        static DataSourceManager& instance();

        bool add(DataSource* dataSource);
        bool erase(DataSource* dataSource);

        unsigned int numberOfActiveDataSources() const;
        DataSource* get(unsigned int index) const;

    private:
        DataSourceManager();
        ~DataSourceManager();

        // prevent copy constructor and copy assignment
        DataSourceManager(const DataSourceManager& dsManager);
        DataSourceManager& operator = (const DataSourceManager& dsManager);

        static DataSourceManager* singletonInstance;

        typedef map<DataSource*,NodeType> Storage;
        Storage mStorage;

        //...

};
```

Neben den nötigen Konstrukten, die die Singleton Eigenschaften implementieren, hier eine kurze Übersicht über andere wesentliche Routinen:

`bool add(DataSource* dataSource);`
`bool erase(DataSource* dataSource);`
Mit Hilfe dieser Funktionen läßt sich dem DataSourceManager eine neue Datenquelle hinzufügen bzw. eine vorhandene löschen. Wichtig beim Löschen ist es, dass zuvor alle mit der

Datenquelle verbundenen Visualisierungsmethoden auch gelöscht werden müssen, da sonst Visualisierungsmethoden im System exisitieren würden, die keine zugehörige Datengrundlage mehr hätten.

`unsigned int numberOfActiveDataSources() const;`
Hierüber kann man die Anzahl der aktiven Datenquellen oder auch Datenfelder erfragen.

`DataSource* get(unsigned int index) const;`
Mit dieser Funktion kann man sich eine Referenz auf eine spezielle Datenquelle übergeben lassen. Über diese Referenz kann man im Anschluß auf die Datenquelle zugreifen.

### 6.7.1 DataSource

Die Datenquellen werden in der `DataSource` genannten Klasse gespeichert. Sie kapselt alle Funktionen, die den Threadsafen Zugriff auf die Daten erlauben. Zusätzlich enthält die Klasse noch einige Dienstfunktionen, die den Umgang mit den Daten erleichtern und Auskunft über deren Typ liefern.

---

**Die *DataSource* Klasse**

---

```
class DataSource
{
    public:
        DataSource(const BoundingBox& boundingBox);
        virtual ~DataSource();

        virtual string getName() const = 0;
        const BoundingBox& getBoundingBox() const;

        void beginEdit() const;
        void endEdit() const;

        void beginRead() const;
        void endRead() const;

    protected:
        BoundingBox mBoundingBox;
        mutable QMutex mEditMutex;
        mutable QMutex mReadMutex;
        mutable int mReaders;

    //...

};
```

Die internen Mutexe regeln den Zuriff auf die Datenquelle. Sowohl für den lesenden als auch für den schreibenden Zugriff wird ein entsprechender Mutex angelegt.

`virtual string getName() const = 0;`
Diese Funtionen gibt den Namen des Datenfeldes zurück, falls ihm ein Name zugewiesen wurde. Ein solcher Name wäre beispielsweise „Temperatur“ oder „Druck“. Neben dem Namen des Feldes werden auch Funktionen benötigt, über z.B. die zugehörige physikalische Einheit (°C, km/h, ...) abgefragt werden kann.

`const BoundingBox& getBoundingBox() const;`
Diese Funktion liefert die BoundingBox der Daten, d.h. den kleinsten Achsen parallelen Quader, der die Datenmenge umfaßt.

```
void beginEdit() const;
void endEdit() const;
void beginRead() const;
void endRead() const;
```
Über diese Funktionen kann sich ein anderes Modul / Probe im System zum lesenden bzw. schreibenden Zugriff auf die Daten anmelden. Wichtig für die Threadsafe Arbeit mit den Daten ist, dass dies gegebenenfalls unter gegenseitigem Ausschluss erfolgen muss, um Dateninkonsistenzen zu vermeiden (siehe Kapitel 5.4.3). Hierfür sind die entsprechenden Mutexe zuständig, die beim Eintritt in die zugehörigen Funktionen automatisch reserviert werden, um den Zustand eines passiven Wartens auf die geforderte Berechtigung zu erzeugen.

Abbildung 6.7 veranschaulicht noch einmal schematisch den Zusammenhang zwischen den einzelnen Klassen der Datenhaltung.

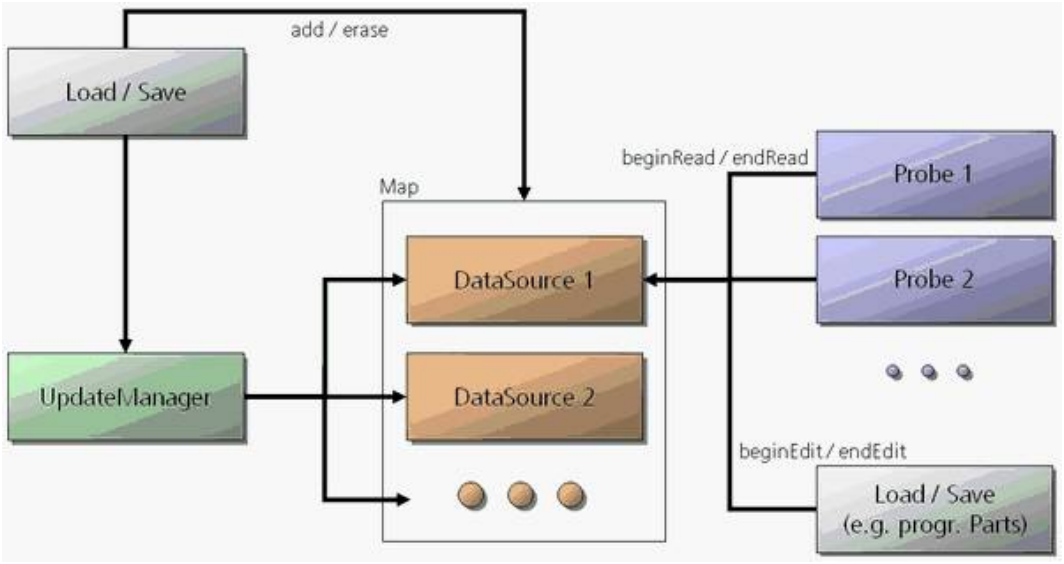



***Abbildung 6.7:*** *Der Zugriff auf die DataSourceManager Klasse und die einzelnen DataSources.*

## 6.8 Grafische Benutzeroberfläche

Wie im Konzept entworfen, wird die grafische Benutzeroberfläche (Graphical User Interface, kurz GUI) als eigenständiges Modul angelegt. So besteht die Möglichkeit, sie schnell gegen eine andere auszutauschen und/oder unabhängig vom Rest des Systems anzupassen. Die GUI besteht aus verschiedenen Teilen, die in Abbildung 6.8 dargestellt sind.

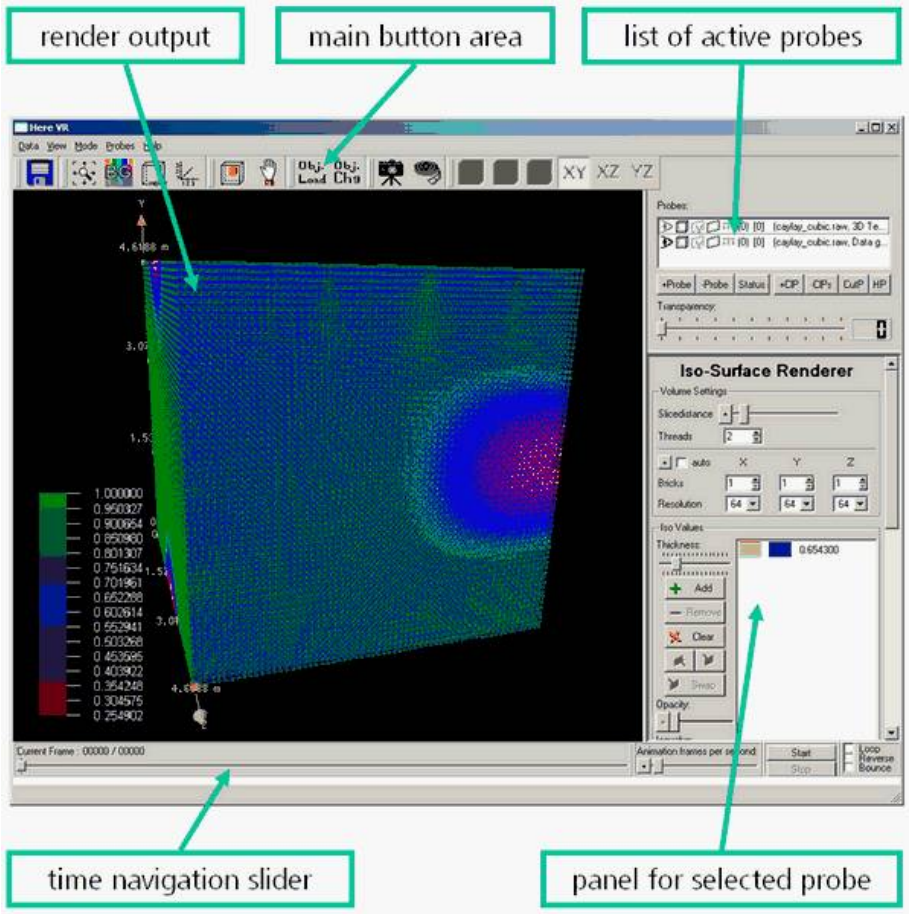



***Abbildung 6.8:*** *Das Hauptfenster und die wesentlichen Bestandteile der grafischen Benutzeroberfläche.*

Die grafische Benutzeroberfläche des Hauptfensters zerfällt auf den ersten Blick erkennbar in 5 Bereiche. Neben diesen Bereichen wird an dieser Stelle auch noch auf einzelne wichtige Unterfenster eingegangen, die sich bei der Benutzung des Programmes dem Anwender präsentieren, wenn er z.B. einen neuen Datensatz lädt oder eine Visualisierungsmethode mit den Daten verknüpft. Die Unterfenster, die für die jeweils aktive Probe alle wichtigen Funktionen abbilden, werden hierbei *Panels* genannt. Sie werden in einem extra Kapitel behandelt.

Sobald der Anwender mit der GUI interagiert, indem er z.B. ein Button anklickt, eine Zahl in ein dafür vorgesehenes Feld einträgt oder mit der Maus die Szene bewegt, wird eine entsprechende Action / Event generiert und an den Central Scheduler zur Auswertung geschickt. Dieser leitet das dann an das betroffene Element weiter, z.B. in dem ein UpdateRunner für

eine Probe angelegt wird, die daraufhin eine Neuberechnung starten muss.

### 6.8.1 Main Button Area


***Abbildung 6.9:*** *Das Hauptmenü und die zugehörigen Buttons.*

Dieser Teil der grafischen Benutzeroberfläche bildet die wichtigsten Menüfunktionen zusätzlich noch einmal auf leicht zugänglichen Buttons ab. Zu den wesentlichen Funktionen, die der Anwendern in diesem Bereich aktivieren kann, gehören:

- Datensatz laden / löschen
- Resize / Reset View
- Hintergrundfarbe wählen
- Manipulationsmodus für Clip / Cutplanes aktivieren
- 3D Umgebunsgzene laden
- Screenshot / Video abspeichern
- Koordinatensystem / Probenumrandung ein- und ausblenden

Die verschiedenen Unterfenster zu diesen Funktionen sind in den Abbildungen 6.10, 6.12, 6.13, 6.14, 6.11 und 6.15 kurz dargestellt.

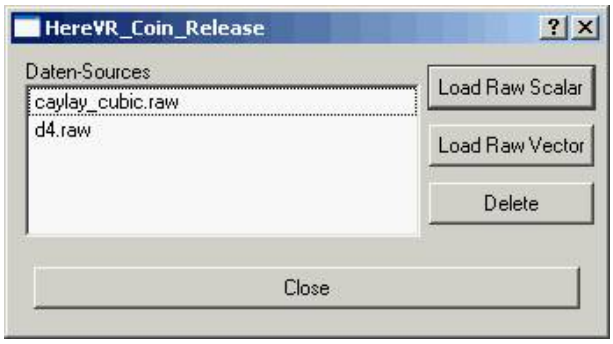



***Abbildung 6.10:*** *Das Lade / Speicherfenster. Im linken Teil des Fensters werden die bereits geladenen Datensätze aufgelistet. Rechts hat der Nutzer die Möglichkeit Skalar- oder Vektorfelder zu laden bzw. diese zu löschen. Wird ein Datensatz gelöscht, so werden automatisch auch alle mit ihm verbundenen Proben aus dem System entfernt.*


***Abbildung 6.11:*** *In diesem Bereich kann der Anwender eine Animation der Darstellung starten. Mit dem großen Schieberegler links können gezielt einzelne Zeitschritte der Datenquelle angefahren werden. Zusätzlich kann noch die Animationsgeschwindigkeit geregelt werden und ob sie z.B. in einer Schleife ablaufen soll.*

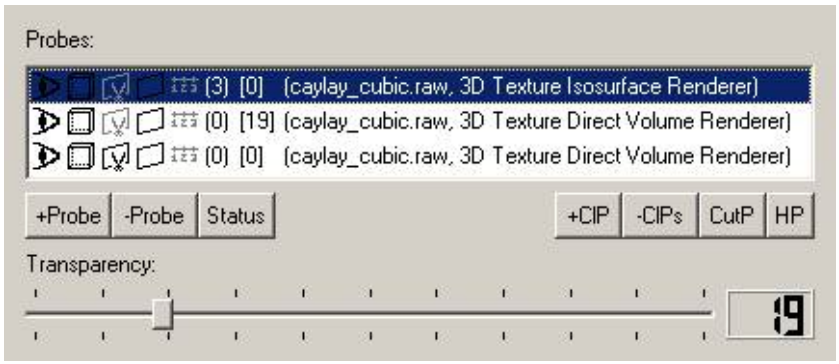



***Abbildung 6.12:*** *Hier werden die aktiven Proben aufgelistet. Neben dem Namen der Visualisierungsmethode wird auch die verknüpfte Datenquelle aufgeführt. Zusätzlich kann der Benutzer sehen, ob die Probe ausgeblendet wurde, ihre Umrandung sichtbar ist, eine Cutplante auf sie angewendet wurde und wie viele Clipplanes mit ihr verknüpft sind. Schließlich wird auch ihr gegenwärtiger globaler Transparenzwert angezeigt, der mit einem Schieberegler im unteren Bereich des Fensters direkt eingestellt werden kann. Mit den +/- Probe Buttons können neue Proben der Liste hinzugefügt bzw. vorhandene entfernt werden.*

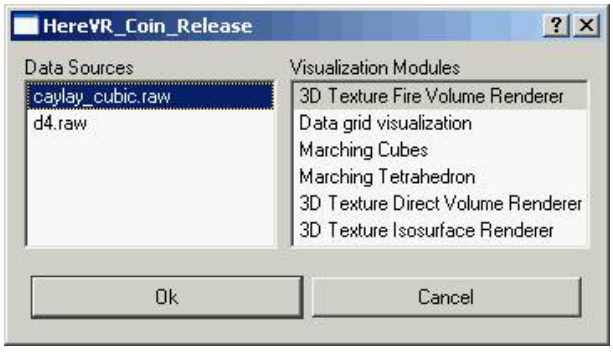



***Abbildung 6.13:*** *Entscheidet sich der Anwender, eine neue Probe zu aktivieren, so wird ihm dieses Fenster präsentiert. Auf der linke Seite werden die zur Verfügung stehenden Datenquellen gelistet. Nachdem dort eine Datenquelle selektiert wurde, werden auf der rechten Seite die für diesen Typ (z.B. Skalar- oder Vektorfeld) unterstützten Visualisierungsmethoden aufgeführt. Hier werden Visualisierungsmethoden, die zwar zum Typ passen würden, vom System aber (z.B. durch fehlenden Graphik Extension Support) nicht unterstützt werden, ausgeblendet.*

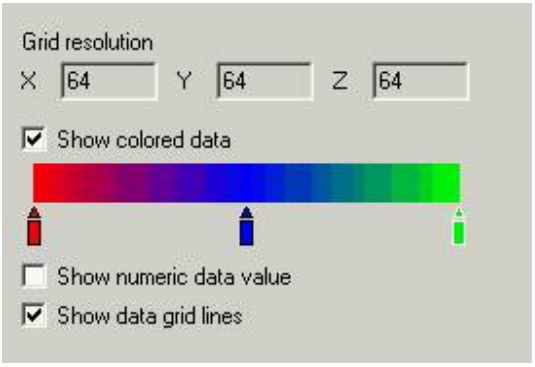



***Abbildung 6.14:*** *In diesem Teil des Hauptfensters werden die Panels der in der Probenliste selektierten Probe angezeigt. Im Beispiel ist es ein Panel zu einer Gitterdarstellung der Datenquelle.*

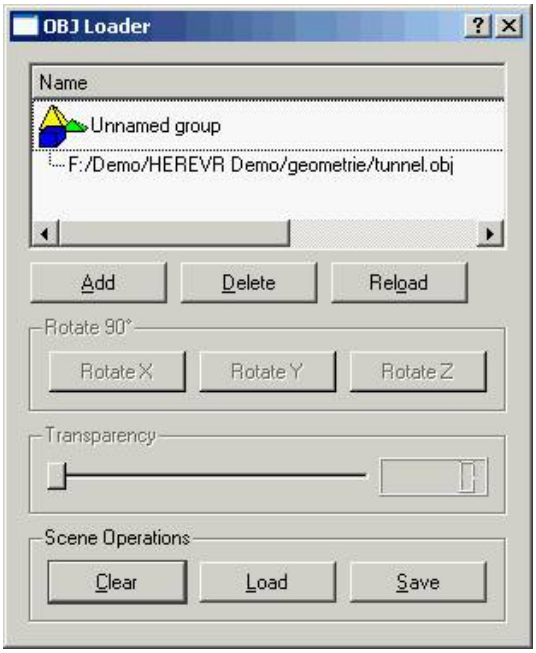



***Abbildung 6.15:*** *Mit Hilfe dieses Fensters können Umgebungsgeometrien zur zusätzlichen Darstellung geladen werden. Hierbei werden diese Geometrien in einer „Szene“ abgelegt, die auf Wunsch auch geladen und gespeichert werden kann. Die Szene „merkt“ sich, welche Objekte wie im Raum arrangiert sind. So können bestimmte Konfigurationen im Nachhinein schnell wieder hergestellt werden. Zusätzlich kann man jedes Geometrie Objekt mit einem Transparenzwert versehen, um es in der Darstellung durchscheinend zu machen bzw. auszublenden.*

### 6.8.2 Panels und zugehörige Factory

Mit Panels werden spezielle zu dedizierten Visualisierungsmethoden zugehörige Fenster bezeichnet. In ihnen werden die Parameter abgebildet bzw. deren Steuerung zugänglich gemacht, die zur jeweiligen Visualisierungsmethode gehören. Da sich die Parameter der einzelnen Visualisierungsmethoden deutlich voneinander unterscheiden, genau wie die Visualisierungsmethoden selbst, ist dieser Teil des Systems sehr flexibel umgesetzt. Über eine Factory registrieren die Visualisierungsmethoden ihre Panels. Zur Laufzeit können diese Panels dann dort bei Bedarf erzeugt werden. Wird eine Visualisierungsmethode aktiviert, so fordert das Hauptfenster das zugehörige Panel bei der Panelfactory an und platziert es in dem dafür vorgesehenen Bereich im GUI.

Zunächst wird eine allgemeine Panel Klasse benötigt, die hier kurz dargestellt ist:

**Die *VisModulePanel* Klasse**

```
class VisModulePanel : public virtual QWidget
{
    Q_OBJECT

    public:
        VisModulePanel(QWidget* parent = 0, const char* name = 0);
        virtual ~VisModulePanel();

        void setProbe(Probe* probe)  mProbe = probe;
        Probe* getProbe() const  return mProbe;

    protected:
        Probe* mProbe;

    //...
};
```

Die Panel Klasse kennt ihre zugehörige Probe und kann diese auch auf Anfrage zurückliefern. Zusätzlich erbt die Klasse alle Eigenschaften von QWidget, der grundlegenden QT Fensterklasse. Panels, die in das System integriert werden, erben von dieser allgemeinen Panelklasse und überladen dabei die definierten Funktionen.

Die zugehörige Factory wird wieder als Singleton angelegt, um mehrfache Instanziierung zur Laufzeit auszuschließen. Genau wie schon bei der `VisModuleFactory` wird hier eine Map zur Speicherung der zur Verfügung stehenden Panels verwendet. Der ganze Zusammenhang ist noch einmal in Abbildung 6.16 verdeutlicht.

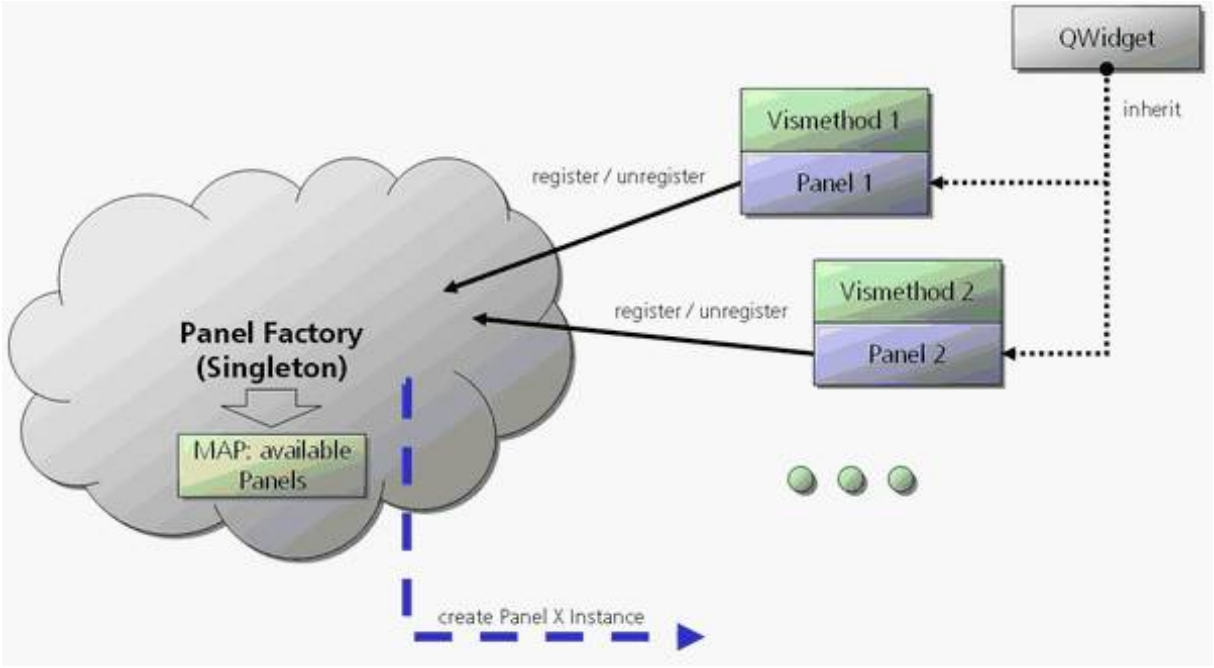



***Abbildung 6.16:*** *Die Panel Factory. Über sie werden die Panels zu den Visualisierungsmethoden erzeugt.*

### 6.8.3 Steuerung

Wichtig für die Realisierung der Steuerung war das korrekte Zusammenspiel zwischen 2D Mauseingabe und 6D Spacemouse Eingabe. Hierbei ist zu beachten, dass die 2D Maus wesentlich weniger Informationen zur Navigation bereitstellen kann, als es die 6D Spacemaus tut. Allerdings kann man sich behelfen, indem man zusätzlich die Maustasten zur Navigation verwendet, um z.B. zwischen Rotation und Translation umzuschalten.

Um ein Objekt im Raum um seinen eigenen Mittelpunkt rotieren zu lassen, verschiebt man das Objekt in den Mittelpunkt des Koordinatensystems, dreht es und verschiebt es wieder zurück. Dieses Prinzip wird auch hier angewandt. Um eine intuitive Steuerung zu ermöglichen, müssen hierbei allerdings die globalen Transformationen beachtet werden.

Die Lage im Raum kann wie erwähnt auf zwei Arten verändert werden: Einmal durch eine 6D Spacemouse und zum anderen durch die normale 2D Maus. In beiden Fällen wird aus den Eingangssignalen der Geräte eine lokale Transformationsmatrix $T_L$ errechnet, die dann mit dem Transformationsknoten des z.Zt. aktiven Objekts verknüpft wird.

Die lokalen Transformationsmatrizen errechnen sich bei einer 2D Maus wie folgt, wobei $d_x$ und $d_y$ der Differenz zwischen aktueller und alter Mausposition entsprechen und $f_1$ und $f_2$ Faktoren sind, um eine gleiche Bewegungsgeschwindigkeit zu erzielen:

$$\text{X/Y-Translation: } T_{L_{xy}} = \begin{bmatrix} 1 & 0 & 0 & d_x \cdot f_2 \\ 0 & 1 & 0 & -d_y \cdot f_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Z-Translation: } T_{L_z} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_y \cdot f_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Rotationswinkel: } \alpha = \sqrt{d_x^2 + d_y^2} \cdot f_2 \cdot 0,5,$$

$$\text{Länge: } l = \sqrt{d_x^2 + d_y^2},$$

$$\text{Faktor: } b = \frac{\sin(\alpha)}{l}$$

$$\text{Quaternion [MH98]: } q = d_y \cdot b + i \cdot d_x \cdot b + j \cdot 0 + k \cdot \cos(\alpha)$$

$$\text{Das Quaternion } q \text{ in Matrixschreibweise: } T_{L_r} = \begin{bmatrix} d_y \cdot b & -d_x \cdot b & \cos(\alpha) & -d_y \cdot b \\ d_x \cdot b & d_y \cdot b & -d_y \cdot b & -\cos(\alpha) \\ -\cos(\alpha) & 0 & d_y \cdot b & -d_x \cdot b \\ 0 & \cos(\alpha) & d_x \cdot b & d_y \cdot b \end{bmatrix}$$

Wenn als Eingabegerät die 6D Spacemouse gewählt wird, so errechnen sich die Transformationsmatrizen aus den Eingangssignalen $t_x$, $t_y$ und $t_z$ für eine Änderung der Translation und $r_x$, $r_y$ und $r_z$ für eine Änderung der Rotation, wie folgt:

$$\text{Translation: } T_{L_T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_y \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{X-Rotation: } T_{L_{rx}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(r_x) & 0 \\ 0 & \sin(r_x) & \cos(r_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Y-Rotation: } T_{L_{ry}} = \begin{bmatrix} \cos(r_y) & 0 & \sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(r_y) & 0 & \cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Z-Rotation: } T_{L_{rz}} = \begin{bmatrix} \cos(r_z) & -\sin(r_z) & 0 & 0 \\ \sin(r_z) & \cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Bei der Verknüpfung mit den aktiven Objekt sind auch noch folgende Matrizen beteiligt:

- Die **globale Transformationsmatrix** $T_G$, die die Lage des Objekts im Raum beschreibt.
- Die **Kamera-Transformationsmatrix** $T_K$, die die Lage der Kamera im Raum beschreibt. Sie wird benötigt, damit eine Mausbewegung nach rechts auch einer Objektbewegung nach rechts entspricht, auch wenn die Szene z.B. um 180° gedreht ist.

Die resultierende Transformationsmatrix $M$ lautet:

$$M = M \cdot T_K \cdot T_G \cdot T_L \cdot T_G^{-1} \cdot T_K^{-1}$$

Um die einzelnen Zusammenhänge besser verstehen zu können, empfiehlt es sich, trotz höherem Rechenaufwand, die Transformationsmatrix $T_L$ in eine Translationsmatrix $T_{L_t}$ und Rotationsmatrix $T_{L_r}$ aufzuteilen. Bei obiger Formel wird außerdem noch nicht um den lokalen Mittelpunkt des Objekts rotiert. Deshalb muss vor der Rotation das Objekt noch in den Ursprung verschoben werden ($T_t$). Die sich daraus ergebende Transformationsmatrix $M$ lautet:

$$M = T_t^{-1} \cdot M_r \cdot T_{G_r} \cdot T_{K_r} \cdot T_{L_r} \cdot T_{K_r}^{-1} \cdot T_{G_r}^{-1} \cdot M_r^{-1} \cdot T_t \cdot M \cdot T_K \cdot T_G \cdot T_{L_t} \cdot T_G^{-1} \cdot T_K^{-1}$$

## 6.9 Szenegraph Abstraktion

Alle Daten, die den Szenegraph betreffen, werden in gesonderte Klassen gekapselt. So kann der Rest des Systems unabhängig von diesem Teil realisiert werden. Wird der Wechsel des Szenegraphens nötig, beispielsweise durch Plattformwechsel von Windows auf Linux oder für die Integration in ein vorgegebenes Visualisierungssystem mit einem bestimmten Zielszenegraphen, so fallen dadurch lediglich Änderungen in diesen gesonderten Teilen an.

Bei der Implementierung wird hierfür eine allgemeine Klasse für den Szenegraphknotentyp mit dem Namen `NodeType` eingeführt. Immer wenn Szenegraphknoten im System übergeben werden müssen, so wird diese Klasse verwendet. Das ist beispielsweise der Fall, wenn eine Visualisierungsmethode dazu aufgefordert wird, aus ihren Ergebnisdaten einen zugehörigen Szenegraphknoten zu generieren und dem UpdateManager zu übergeben.

Insbesondere bei der in Kapitel 6.8.3 beschriebenen Methode, durch die Szene zu navigieren, wird es nötig, neben der reinen Abstraktion des Szenegraph Knotentypens auch die gesamte Ansicht auf die Szene zu kapseln. So können entsprechende Matrizen bzw. Kameraeinstellungen über eine generalisierte Schnittstelle eingesteuert werden, ohne sich um die tatsächlich dahinter verborgenen Objekte (z.B. Rotations- oder Translationsknoten im Ergebnis-Szenegraph) kümmern zu müssen. In der Realisierung geschieht das mit Hilfe der Klasse `SceneView`. Sie verwaltet den Szenegraph des Systems mit den dazugehörigen Kamera / Lichteinstellungen. Ferner stellt sie Funktionen für das Hinzufügen / Entfernen von Clip- und Cutplanes bereit. Um Mehrfachinstanziierungen zu verhindern, wird auch sie als Singleton angelegt.

## 6.10 Zusammenfassung

Im vorliegenden Kapitel werden die wichtigen Klassen und Funktionalitäten eines Visualisierungssystems für Strömungsdaten vorgestellt. Mit der Definition des Probepools und der zugehörigen Factory wird der eigentliche Kern des Systems entwickelt, welcher die Grundlage für eine parallele Abarbeitung der Visualisierungsdaten darstellt. Mit Hilfe dieser Werkzeuge und der auf Vererbung basierenden Klassenhierarchie für die unterschiedlichen Visualisierungsmethoden ist es möglich, die in den vorangegangenen Kapiteln aufgestellten Anforderungen zu erfüllen und die im Konzept definierten Ansätze zu realisieren.

Das Konzept der generischen Visualisierungsmethode wird spezifiziert. Sie dient einer allgemeinen abstrakten Beschreibung einer Visualisierung auf Basis standardisierter Schnittstellen. Zu ihrer Steuerung wird das Konzept der Actions spezifiziert und die eventbasierte Kommunikation umgesetzt.

Als zentrales Element wird der Update Manager vorgestellt, der zusammen mit den Updaterunnern das Konzept des Central Schedulers umsetzt. Er repräsentiert die zentrale Instanz des Systems, über die alle Steuervorgänge laufen. Zusätzlich koordiniert und überwacht er die einzelnen Visualisierungsmethoden und regelt die Abläufe im System für die graphische Darstellung, die mit Hilfe einer Szenegraph Abstraktion realisiert wird.

Zur Kapselung der Datenhaltung wird der Datamanager spezifiziert, der die systemweite Aufgabe der Strömungsdatenspeicherung übernimmt.

Bei der Spezifikation des Systems wird darauf geachtet, die einzelnen Module über abstrakte Schnittstellen weitestgehend unabhängig und parallelisiert voneinander zu realisieren und deren Aufgabe und Funktion möglichst einfach und übersichtlich zu halten.

# Kapitel 7

# Umsetzung und Beispiele

Zu Beginn dieses Kapitels werden einige der Visualisierungsmethoden vorgestellt, die im Zuge der Realisierung dieses massiv parallelen Visualisierungssystems für große Strömungsdatenmengen umgesetzt wurden. Exemplarische Beispiele, die sich durch neue Ideen und Lösungswege hervorheben, die während der Umsetzung dieser Arbeit erschlossen werden konnten, werden herausgegriffen und ihre wichtigsten Grundsätze bzw. ihr prinzipieller Aufbau kurz erläutert.

Das Visualisierungssystem, das durch die Ideen und Konzepte dieser Arbeit realisiert wurde, kam im Laufe seiner Entstehung in mehreren Forschungsprojekten zum Einsatz, die am Fraunhofer Institut für Graphische Datenverarbeitung durchgeführt wurden. Um das System an die jeweiligen Gegebenheiten anzupassen, wurden die im Konzept vorgestellten Mittel genutzt, die z.B. den Wechsel der Plattform oder den Austausch des graphischen Ausgabeformates („Ziel-Szenegraph“) zulassen. Auf diese Weise konnte das System erfolgreich in unterschiedliche Umgebungen migriert und dort zur Problemlösung angewandt werden. Gegen Ende dieses Kapitels werden die entsprechenden an Projekte gebundenen Beispiele kurz skizziert.

## 7.1 Visualisierungsmethoden

Ein wesentliches Kriterium, wie leistungsfähig und flexibel ein Visualisierungssystem für Strömungsdaten ist, zeigt sich in erster Linie darin, wie viele verschiedene Visualisierungsmethoden es dem Nutzer anbieten kann. Sollte sich eine gewünschte Methode nicht unter ihnen befinden, ist es von entscheidender Bedeutung, wie schnell und einfach die entsprechende Fähigkeit nachträglich integriert werden kann.

An dieser Stelle wird auf einzelne spezialisierte Visualisierungsmethoden eingegangen, die im Rahmen dieser Arbeit umgesetzt wurden. Darüber hinaus wurden zahlreiche weitere implementiert, die in Kapitel 7.1.6 kurz aufgelistet sind.

### 7.1.1 Datagrid Visualisierung

Mit die einfachste Visualisierungsmethode, die in das Visualisierungssystem integriert wurde, ist die „Datagrid“ genannte Methode. Sie erzeugt im Datenraum ein äquidistantes Gitter, das sie visualisiert. Die Gittereckpunkte werden anhand einer einstellbaren Farbpalette entsprechend ihres Wertes gefärbt. Auf Wunsch können auch die Zahlenwerte, die an der entsprechenden Stelle aus dem Datenraum ausgelesen werden, an die zugehörigen Gitterpunkte geschrieben werden (Abbildung 7.1).

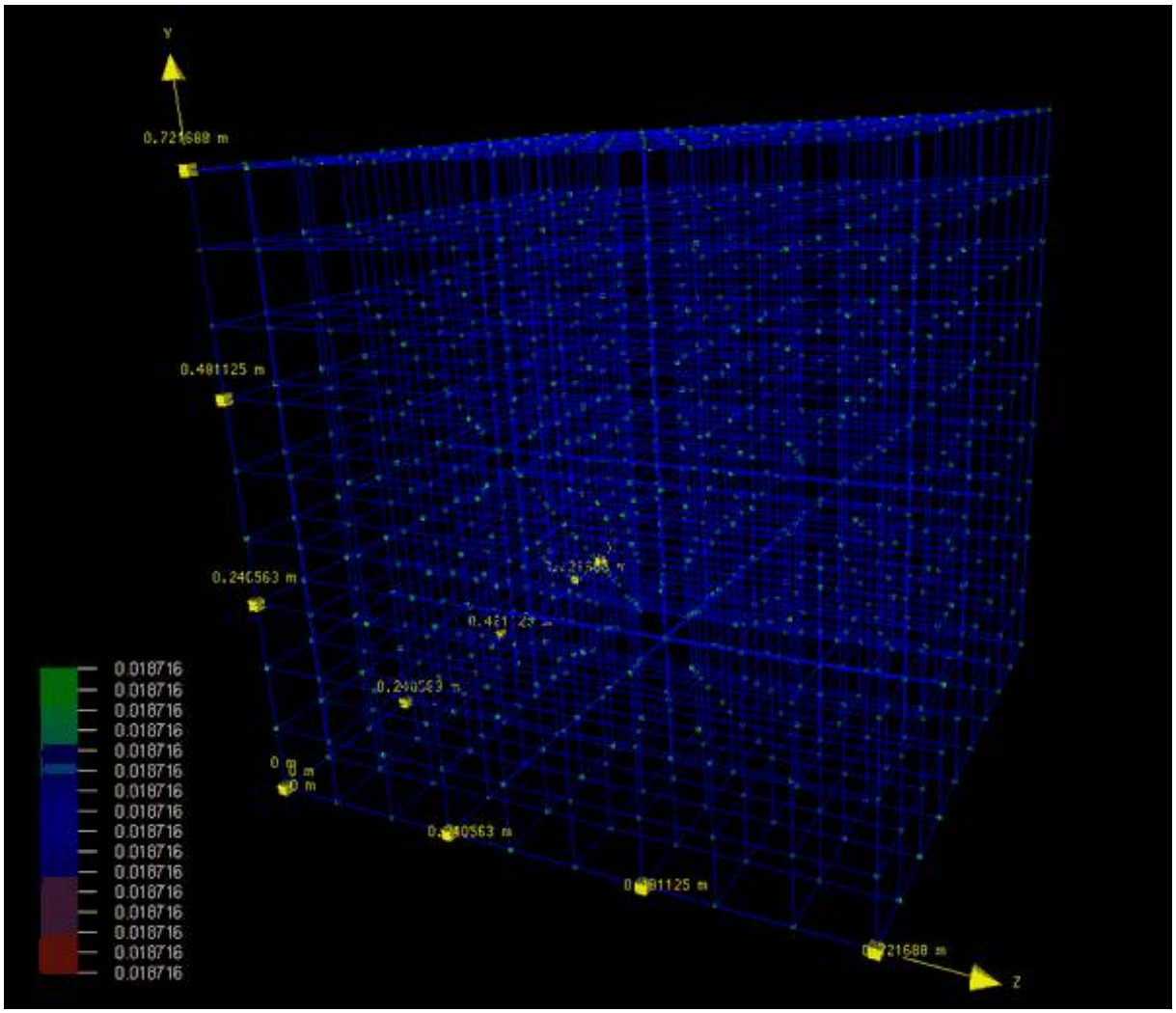



***Abbildung 7.1:*** *Datagrid Visualisierung: In den Datenraum wird ein regelmäßiges Gitter eingezogen. Die Eckpunkte werden anhand des vorgefundenen Wertes und einer einstellbaren Farbpalette eingefärbt. Auf Wunsch werden zusätzlich die aus der Datenquelle eingelesenen Werte auf die Gitterpunkte geschrieben.*

Für große Szenen wird diese Darstellung schnell unübersichtlich. Sie eignet sich vorwiegend für Debug Zwecke, um beispielsweise Fehler oder Inkonistenzen innerhalb der Werte des Datenfeldes zu orten. Außerdem kann man mit ihrer Hilfe z.B. auch schnell interessante Bereiche finden, in denen sich, lokal begrenzt, besonders hohe oder besonders niedrige Werte im Datenraum befinden, sogenannte *Peaks*.

### 7.1.2 Line Integral Convolution

In diesem Unterkapitel wird das sogenannte „Line Integral Convolution“ (LIC) Visualisierungsverfahren kurz vorgestellt und erläutert, wie es im Visualisierungssystem umgesetzt wurde. Das LIC Verfahren beruht im Wesentlichen auf einer Faltung des Wegintegrals.

Die Idee des Line Integral Convolution ist eine von Brian Cabral und Leith Leedom 1993 [CL93] vorgestellte Technik zur Visualisierung von Vektorfeldern. Die Darstellung von Vektorfeldern findet neben wissenschaftlichen Anwendungen auch in Bereichen der Kunst, Bildbearbeitung und Special Effects ihren Einsatz. Das LIC-Verfahren hat als Eingabe ein Vektorfeld und ein zwei-dimensionales Bild (beispielsweise eine Rauschtextur) und erzeugt daraus ein Ausgabebild (Abbildung 7.2).

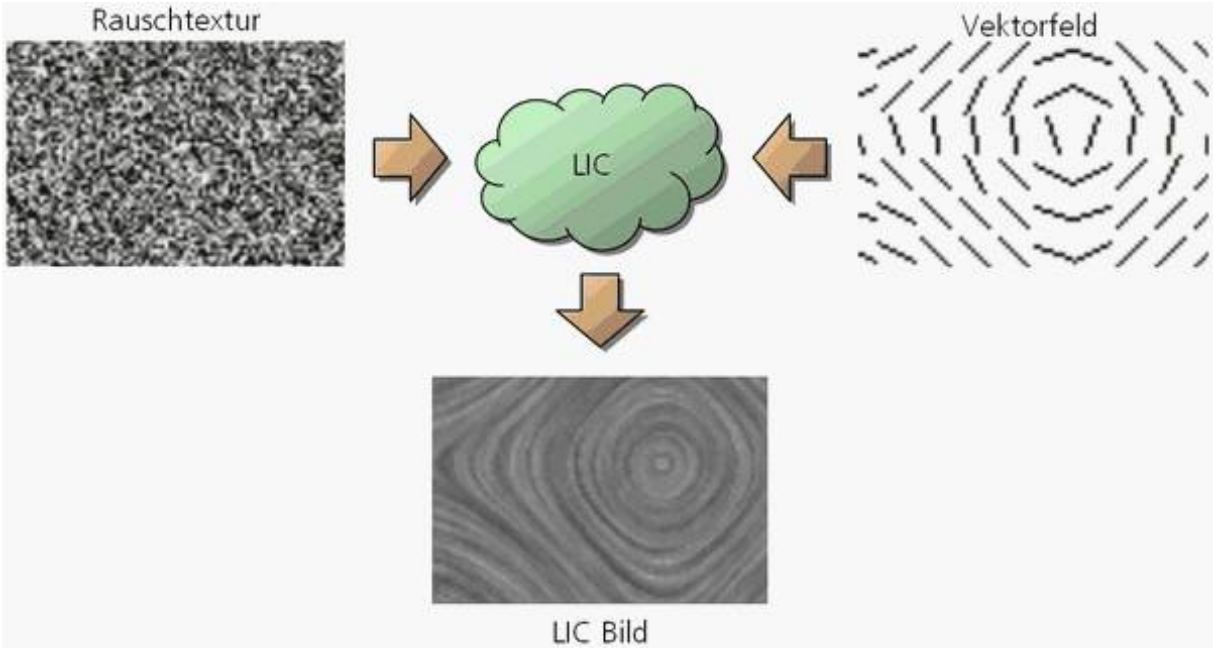



***Abbildung 7.2:*** *Schematische Darstellung des LIC-Verfahrens*

Die LIC Technik selbst dient vor allem der Darstellung von beliebigen zwei-dimensionalen Vektorfeldern, kann aber auch für drei-dimensionale Vektorfelder verwendet werden, indem man sich beispielsweise nur eine Ebene des drei-dimensionalen Vektorfeldes herausgreift (wie in HereVR geschehen) oder den Algorithmus für die dritte Dimension erweitert, wobei hier das Eingabebild (Rauschtextur) natürlich auch drei-dimensional sein muss und die Ausgabe ebenso ein drei-dimensionales Bild ergibt. Der Algorithmus entspricht einem ein-dimensionalen Filter, der auf eine so genannte „Streamline“ (siehe auch Kapitel 7.1.4) angewandt wird. Eine „Streamline“ kann man sich dabei als den Weg vorstellen, den ein massenloser Partikel nehmen würde, wenn man ihn an entsprechender Stelle in das Vektorfeld wirft und er entlang der Strömung „weggetragen“ wird.

#### 7.1.2.1 Faltung

In der Mathematik und insbesondere in der Funktionalanalysis beschreibt die Faltung (englisch: convolution) einen mathematischen Operator, welcher für zwei Funktionen $f$ und $g$ eine dritte Funktion $h$ liefert, die die „Überlappung“ zwischen $f$ und einer gespiegelten verschobenen Version von $g$ angibt:

$$\text{Notation: } h(t) = (f * g)(t)$$

$$\text{Bedeutung: } h(t) = \int f(u)g(t-u)du, \text{ diskretisierter Fall: } \sum_u f(u)g(t-u)$$

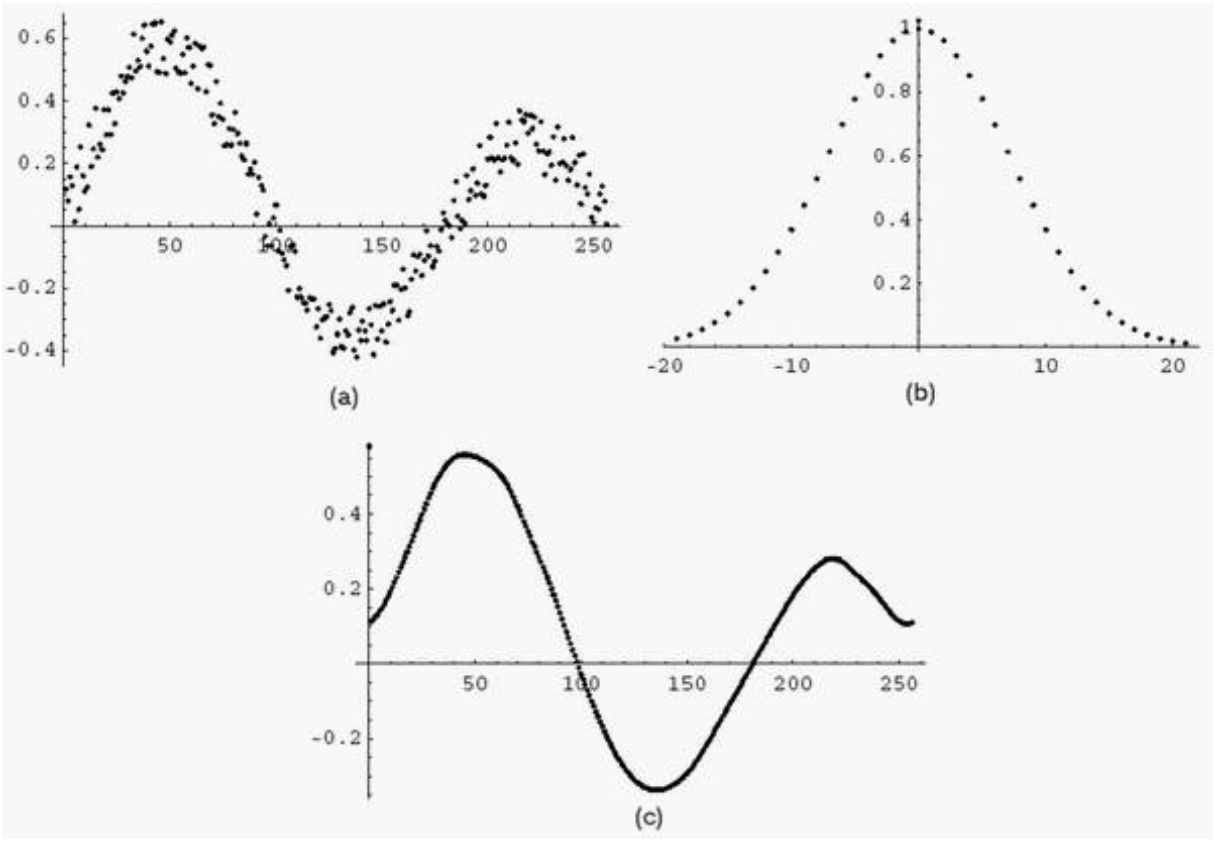



***Abbildung 7.3:*** *Faltung: (a) Verrauschte Ursprungsfunktion, (b) Faltungskern, (c) Gefaltete Funktion [Har]*

In der Bildverarbeitung werden Faltungen zu vielfältigen Zwecken eingesetzt, beispielsweise können in Bildern Kanten mit speziellen Faltungsmatrizen detektiert werden, oder das gesamte Bild kann weichgezeichnet oder geschärft werden. Man spricht in diesem Zusammenhang auch von *Filtern*.

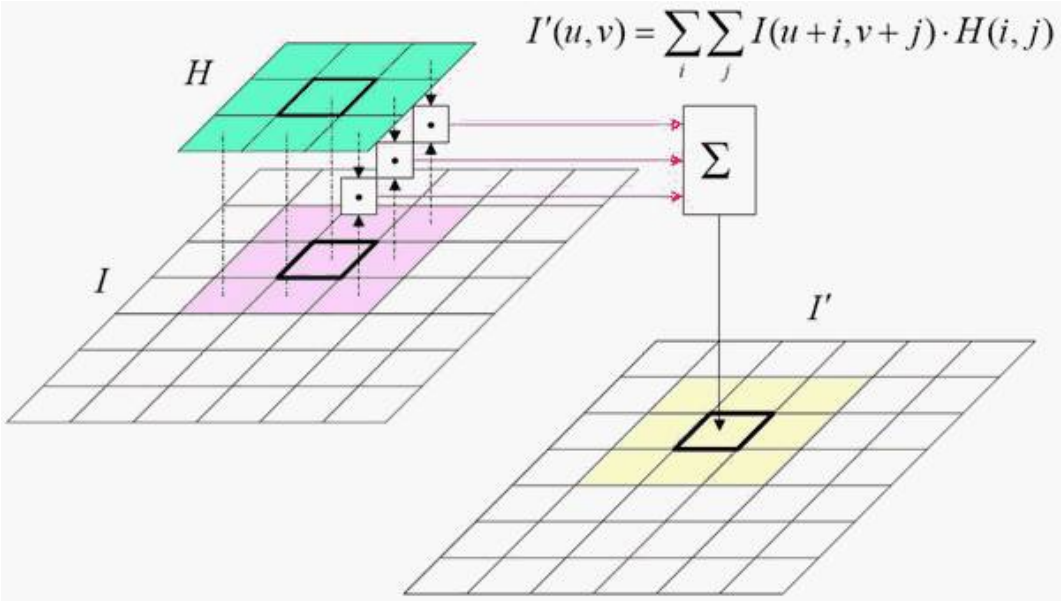



***Abbildung 7.4:*** *Schematische Darstellung einer Faltung [Bur05]*

Anschaulich wird die Faltungsmatrix $H$ über das Bild mit der Bildfunktion $I$ geschoben (siehe Abbildung 7.4), wobei alle umliegenden Pixelwerte, die innerhalb der Faltungsmatrix $H$ liegen, mit den Werten in der Faltungsmatrix multipliziert und aufsummiert werden (siehe auch Abbildung 7.5).

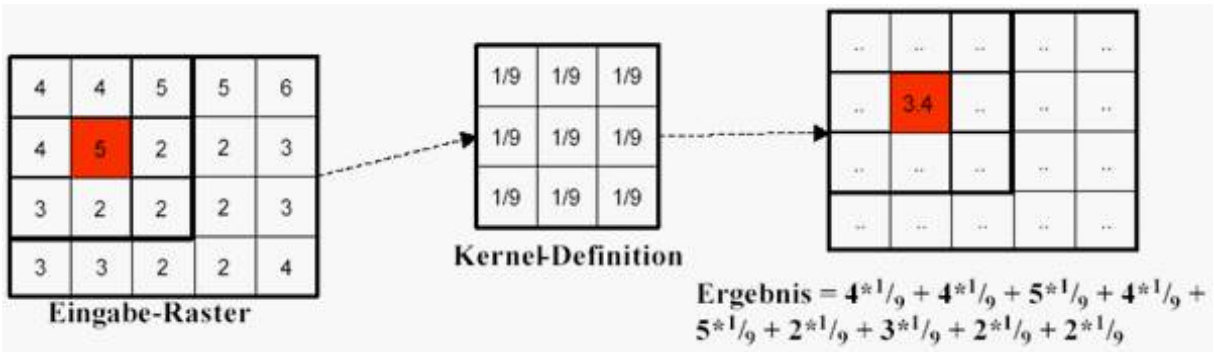



***Abbildung 7.5:*** *Beispielrechnung einer Faltung zur Mittelwertbildung [Lor99]*

#### 7.1.2.2 Wegintegral

Der Begriff des Wegintegrals (Kurvenintegral, englisch: line integral, path integral) bezeichnet zum Beispiel in einem Kraftfeld die Energie, die ein Objekt verbraucht oder gewinnt, wenn es sich in diesem bewegt, bzw. die Arbeit (Arbeit = Kraft * Weg), die bei der Bewegung verrichtet wird.

**Definition 7.1 (Wegintegral)** *Sei die Teilmenge $U \subset \Re^n$ offen, $f : U \mapsto \Re^n$ ein stetiges Vektorfeld, $w : [a, b] \mapsto U$ ein stückweise differenzierbarer Weg und $t_0, \ldots, t_n$ eine Zerlegung von $w$, so dass $w\,[t_i, t_{i+1}]$ stetig differenzierbar ist für $i = 0, \ldots, m-1$. Dann heißt*

$$\int_w f = \sum_{i=0}^{m-1} \int_{t_i}^{t_{i+1}} \langle f(w(t)), w'(t) \rangle \, dt \tag{7.1}$$

*das Wegintegral von $f$ längs $w$. In der Kurzschreibweise auch als $\oint_w f$ notiert.*

Eine Möglichkeit, das Wegintegral anzunähern, ist eine Berechnung durch Riemann-Summen, wie im Beispiel in Abbildung 7.6 deutlich wird. Zunächst wird der Weg (grüne Kurve der Abbildung, welche durch die Funktion k beschrieben ist) durch gerade Liniensegmente (blaue Linie) diskretisiert $(x_k, y_k)$, was ausgehend von einer äquidistanten Zerlegung in X-Richtung geschieht. Dann lässt sich das Wegintegral des Kraftfeldes $f$ durch die Funktion $W$ berechnen.

$$k(x) = \sqrt{1 - x^2} \ , \ x_k = k * \Delta x \ , \ y_k = \sqrt{1 - x_k^2} \ , \ \Delta x = 0.33 \tag{7.2}$$

$$f(x, y) = \begin{bmatrix} x + y \\ x - y \end{bmatrix} \ , \ W = \sum_{k=0}^{2} \begin{bmatrix} x_k + y_k \\ x_k - y_k \end{bmatrix} * \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \ , \ \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} |x_k - x_{k+1}| \\ |y_k - y_{k+1}| \end{bmatrix} \tag{7.3}$$

#### 7.1.2.3 Berechnung

Zur Berechnung eines Ausgabebildes $F'$ der LIC Visualisierungsmethode (F') wird nun die folgende Formel aus [CL93] angewendet:

$$F'(x, y) = \frac{\sum_{i=0}^{l} F(\lfloor P_i \rfloor) h_i + \sum_{i=0}^{l'} F(\lfloor P_i' \rfloor) h_i'}{\sum_{i=0}^{l} h_i + \sum_{i=0}^{l'} h_i'} \tag{7.4}$$

Das Integral des Faltungskerns läßt sich hierbei wie folgt schreiben:

$$h_i = \int_{s_i}^{s_i + \Delta s_i} k(w) dw \ , \ s_0 = 0, s_i = s_{i-1} + \Delta s_{i-1} \tag{7.5}$$

Man folgt der Streamline an der Stelle $(x, y)$ im Eingabebild, das zuvor in beliebiger Abtastung über das Vektorfeld gelegt wird, in positiver und negativer Richtung eine festgelegte

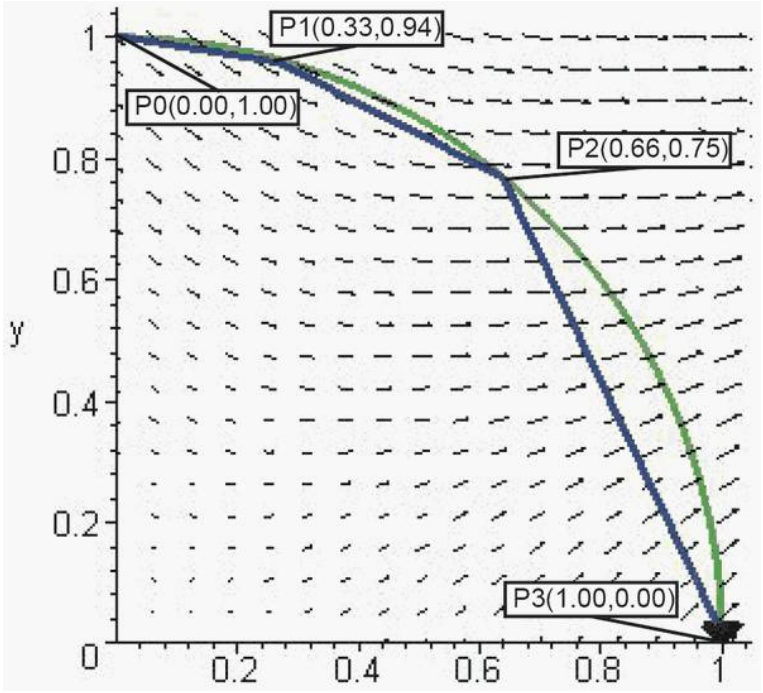



***Abbildung 7.6:*** *Wegintegral eines Vektorfelds.*

Anzahl Felder ($l$ und $l'$), was einer ein-dimensionalen Bildfunktion entspricht ($F$ und $F'$). Die erste Summe im Zähler entspricht dabei dem Weg in positiver Richtung und die zweite Summe dem Weg in negativer Richtung. Diese Bildfunktion faltet man dann mit einem geeigneten Faltungskern $k$, beispielsweise dem Hanning-Filter [han05], der Charakteristiken eines Tiefpass-Filters aufweist, periodisch ist und eine einfache analytische Form besitzt (Der Hanning-Filter bildet die Cosinus-Funktion auf einen Wertebereich zwischen 0 und 1 ab: $f(\theta) = 0.5 + 0.5 * \cos(\theta)$).

Für jedes Liniensegment der Streamline wird dabei das exakte Integral ($h_i$ und $h_{i'}$) des Faltungskerns über die Länge des Liniensegments, die bei jedem Pixel variiert, berechnet. Dieses Integral wird auch zur Normalisierung der Faltung im Denominator der Formel verwendet. Die Normalisierung mit dem Integral des Faltungskerns führt zu einem Ausgabebild mit gleichmäßiger Helligkeit, unabhängig von der Form des verwendeten Filters und der Länge der Streamline.

#### 7.1.2.4 Bump Mapping und LIC

Um in der wissenschaftlichen Darstellung eines Vektorfeldes mittels des LIC-Verfahrens noch mehr Informationen visualisieren zu können, wird auf das berechnete LIC-Bild eine Bump Map (siehe Kapitel 3.6.5) gelegt. Das daraus entstehende Bild wird als „BLIC"-Bild (Bump Mapped LIC-Bild) bezeichnet. Die Einführung der Bump Map erlaubt über das eigentliche LIC hinaus, die Repräsentation einer weiteren Komponente der Daten. Diese wird hierfür als Skalarfeld auf die für das Bump Map Verfahren benötigte Height Map abgebildet. Bei diesen Skalarwerten kann es sich dabei um beliebige Werte handeln. Im Falle von höherdimensionalen Vektorfeldern als das zwei-dimensionale LICVerfahren, bietet sich jede weitere Dimension als Skalarwert zur Berechnung der Bump Map an. Eine weitere sinnvolle Berechnung findet sich in [SM00], bei der die Tendenz der Wirbel bei der Berechnung der Bump Map Einfluss findet.

Für die Realisierung im vorgestellten Visualisierungssystem wurde der Einfachheit halber eine Abbildung gewählt, die den Winkel des anliegenden Vektors zur X-Achse, wie in Abbildung 3.4 ersichtlich, auf Grauwerte abbildet. Die Benutzung der beschriebenen Funktion hat eine Hervorhebung der Wirbel zur Folge 7.7.

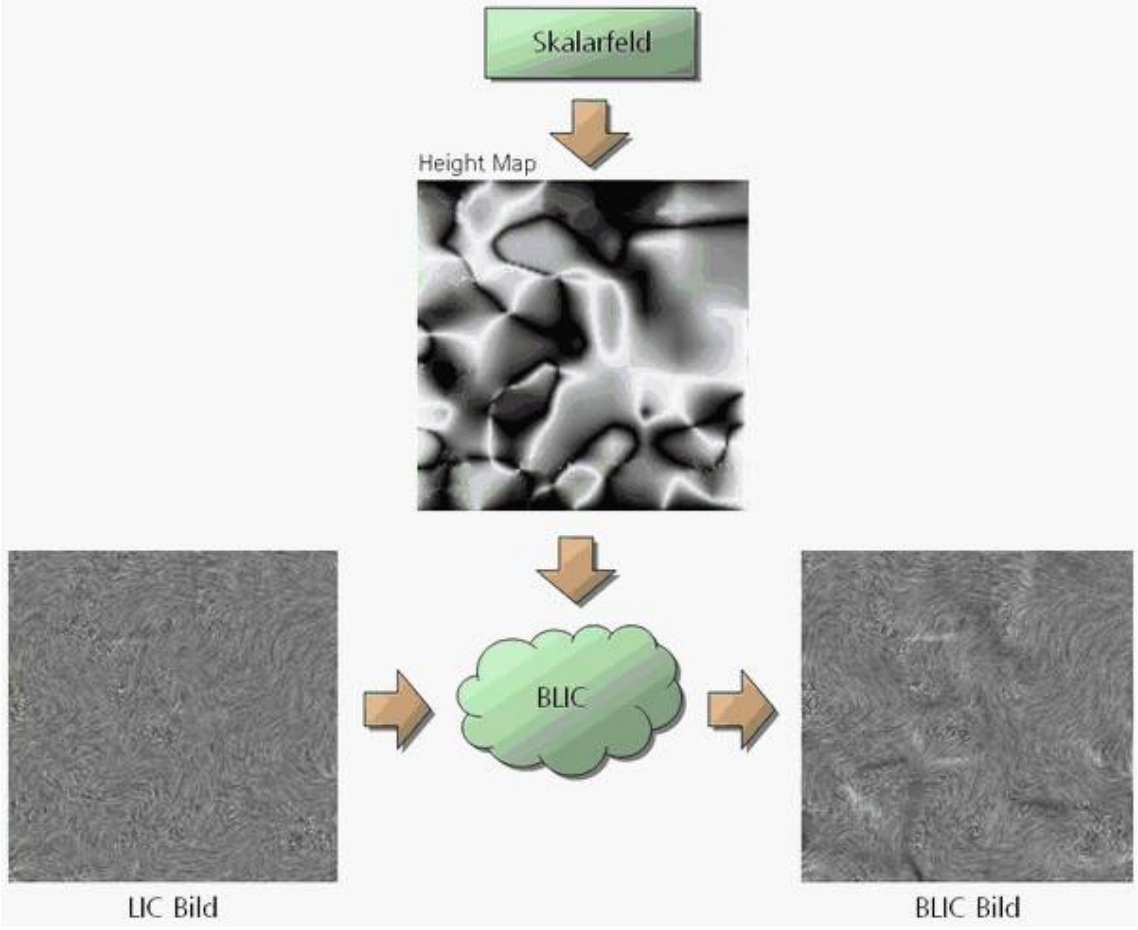



***Abbildung 7.7:*** *BLIC: LIC wird mit Bump Mapping kombiniert.*

#### 7.1.2.5 Parallelisierung

Die eigentliche Parallelisierung des Visualisierungsverfahrens basiert auf der vorhandenen Zerlegung der Textur in X-Richtung, die durch das Abtastgitter gegeben ist. Man könnte aber ebenso die Zerlegung in Y-Richtung als Grundlage nehmen. Die Obergrenze der einstellbaren Threads ist durch die Abtastgröße festgelegt. Ein einzelner Thread übernimmt bei der Parallelisierung die Berechnung der Gitterpunkte und die Berechnung der Verschiebung der Texturkoordinaten der ihm zugeteilten Spalten (Abbildung 7.8). Diese Berechnungen werden an einen von ihm erzeugten Szenegraph Knoten weitergegeben, der für die spätere Darstellung zuständig ist.

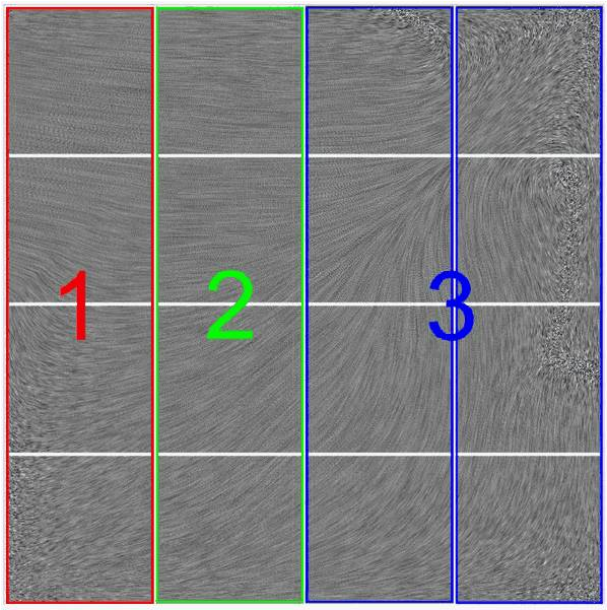



***Abbildung 7.8:*** *Threadverteilung beim LIC-Verfahren.*

#### 7.1.2.6 Panel

In diesem Abschnitt wird kurz das zur Visualisierungsmethode passende Panel mit seinen wichtigsten Funktionen vorgestellt (Abbildung 7.9).

**Textures**

Im Abschnitt „Textures“ wird dem Benutzer in einer Listbox die Möglichkeit zum Laden von Bild-Dateien angeboten, die im Programmpfad gefunden wurden oder durch den „Load Texture“-Button mit Hilfe eines Datei-Öffnen-Dialogs dieser ListBox hinzugefügt und ausgewählt wurden. Nach Auswahl einer Bild-Datei wird diese direkt als LIC-Textur eingesetzt.

**NoiseImage**

Dieser Abschnitt erlaubt die Erzeugung einer Zufallstextur. Dabei kann man entweder

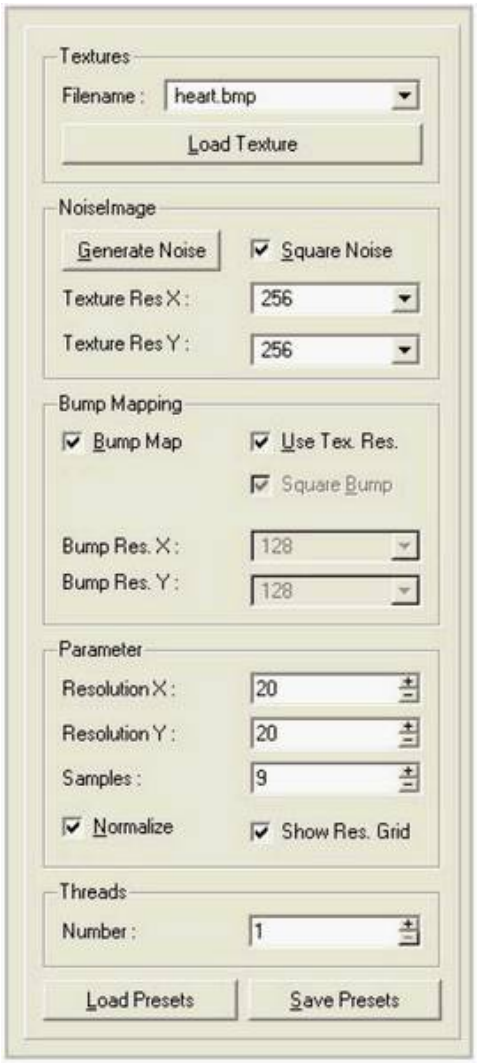



***Abbildung 7.9:*** *Panel zur LIC Visualiesierungsmethode.*

die Auflösung in X- und Y-Richtung getrennt einstellen oder durch die „Square Noise"-Checkbox beide Parameter zugleich variieren.

**Bump Mapping**

Hier lässt sich Bump Mapping aktivieren und deaktivieren, die Auflösung in der gleichen Weise wie im NoiseImage-Abschnitt einstellen, oder die Height Map der Auflösung der gewählten LIC-Textur anpassen.

**Parameter**

In diesem Bereich lassen sich die Parameter des LIC-Verfahrens einstellen: Man kann die Abtast-Auflösung variieren und die zugehörige Gitterdarstellung aktivieren, sowie die Anzahl Verschiebungen der Texturkoordinaten (samples) einstellen, wie auch die Normalisierung des Verschiebevektors vornehmen.

**Threads**
Die Anzahl der Threads mit der das LIC-Verfahren ausgeführt werden soll, lässt sich in diesem Bereich festlegen.

**Presets**
Dem Benutzer wird in diesem Abschnitt die Möglichkeit gegeben, den derzeitigen Zustand aller GUI-Parameter mit Hilfe eines Dialogs in eine Datei zu speichern oder zu laden.

#### 7.1.2.7 Klassenstruktur

Wie in Kapitel 6 beschrieben, werden zur Integration der LIC Visualisierungsmethode in das Gesamtsystem bestimmte vorgegebene Klassen beerbt und deren Funktionen überladen. Hierzu gehören z.B. die in Kapitel 6.4 vorgestellte Vismodule oder auch die in Kapitel 6.5 erläuterte Panel Klasse. In Abbildung 7.10 wird ein schematischer Überblick zur Klassenstruktur des LIC Verfahrens präsentiert.

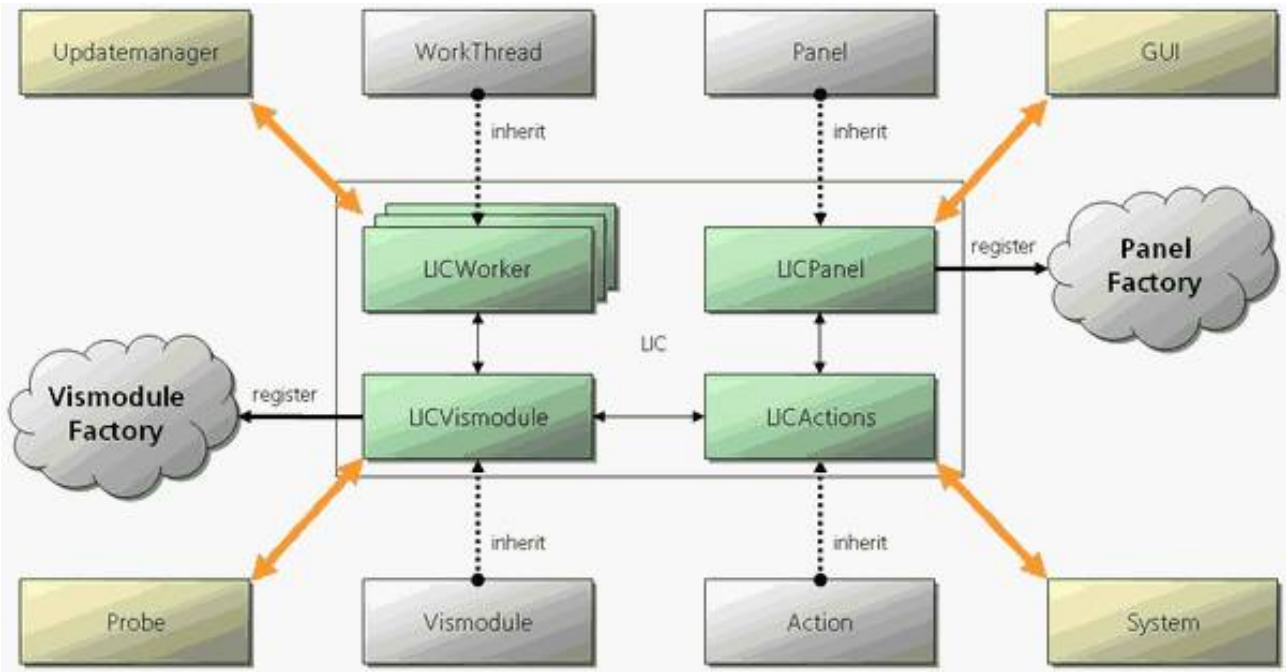
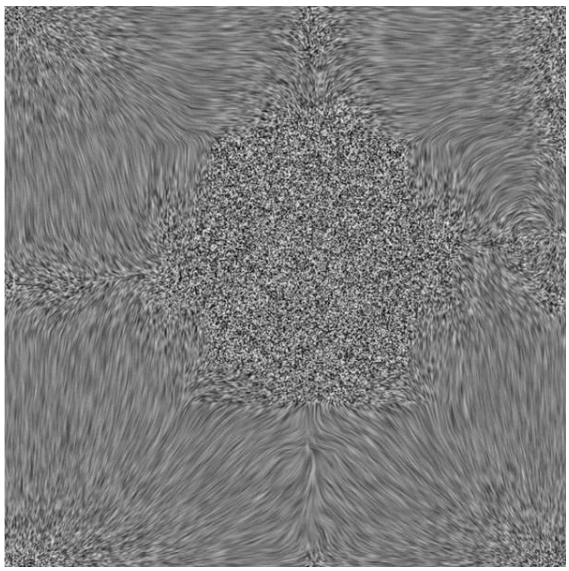


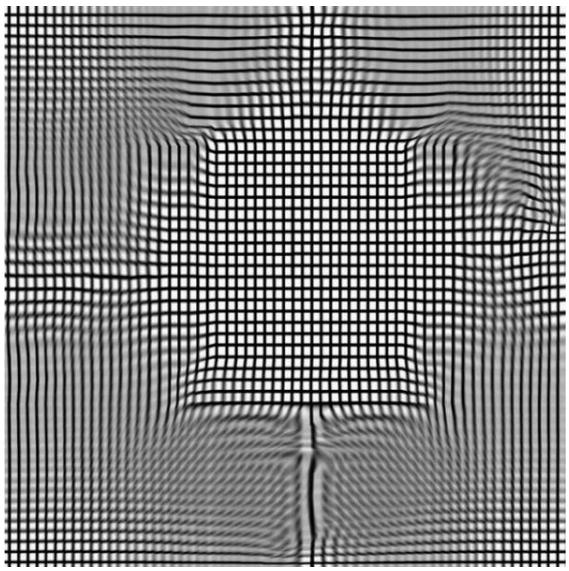

***Abbildung 7.10:*** *Die Klassen, die für die Implementierung des LIC Verfahrens verwendet werden.*

#### 7.1.2.8 Ergebnisbilder

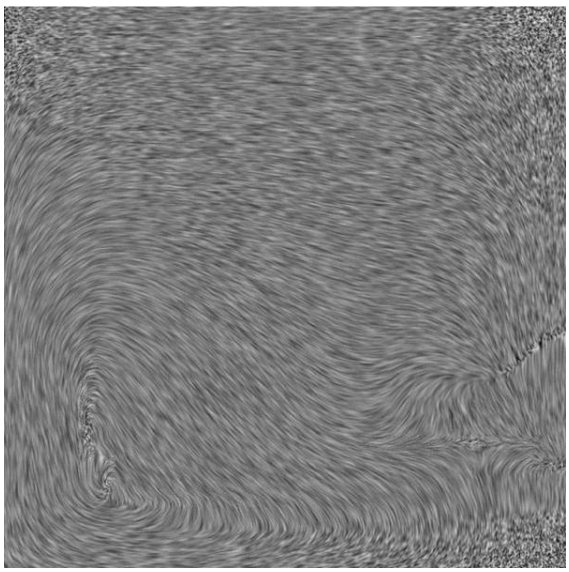
Zum Abschluss werden an dieser Stelle weitere Ergebnisbilder der LIC Visualisierungsmethode präsentiert (Abbildung 7.11).



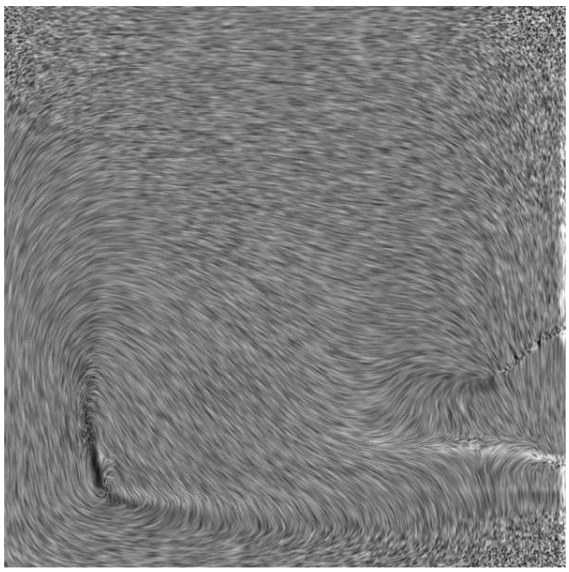
LIC-Verfahren mit Rauschtextur



Dasselbe Strömungsfeld mit einer geladenen Gittertextur



LIC-Verfahren ohne Bump Mapping



LIC-Verfahren mit Bump Mapping

***Abbildung 7.11:*** *Verschiedene Parametereinstellungen und deren Resultate im LIC Verfahren.*

### 7.1.3 Streamlines

Den Pfad, den ein virtuelles massenloses Teilchen bzw. Partikel in einem Strömungsfeld zurücklegt, wird in der Visualisierung als „Strömungslinie“ bezeichnet. Dieses Partikel wird anhand der vorliegenden Strömung Stück für Stück über die Zeit durch das Strömungsfeld bewegt. Verbindet man die dabei erreichten Punkte mit einzelnen Linien, entsteht ein langes Band, welches aus Linienteilstücken besteht. Anstelle von Linien könnte man auch Röhren, Pfeile oder ähnliches einsetzen. Letztendlich bleibt die Realisierung dem Entwickler überlassen und hängt zudem von der Aufgabenstellung und den Anforderungen an das System ab. Dieses Band ist die Strömungslinie.

#### 7.1.3.1 Streamlines, Timelines, Streaklines

Je nach Ansatz muss man hierbei unterschiedliche Typen von Strömungslinien unterscheiden. In statischen Strömungsfeldern benutzt man die sogenannten „Streamlines“ („Strömungslinien“) zur Visualisierung, da für deren Berechnung nur ein Zeitschritt vorliegt. In nichtstatischen Strömungsfeldern wird zwischen Timelines (Zeitlinien) und Streaklines (Streifen-, Spurlinien) unterschieden. Oft wird der Begriff „Streamlines“, der ja nur für einen speziellen Fall dieser Strömungslinien zutreffend ist (statisches Feld), auch zusammenfassend für alle drei Arten gebraucht.

**Streamlines**

Hierbei wird einmalig eine beliebige Anzahl von Teilchen in einem zugrundeliegenden statischen Strömungsfeld erzeugt. Die jeweils zu berechnenden und die zuletzt ermittelten Koordinatenpunkte, die das Teilchen in der Strömung verfolgt, werden mit Linien verbunden bis zum gewünschten Berechnungsschritt bzw. bis zur gewählten Linienlänge. Ändert sich das Strömungsfeld, wird die alte Streamline komplett gelöscht, eine neue erzeugt und in dem neuen Vektorfeld wiederum bis zum x-ten Berechnungsschritt gezeichnet. In einer zeitlich ablaufenden Simulation erhält man dadurch den Eindruck von sich bewegenden „Bändern im Wind“.

In Abbildung 7.12 ist der Pfad eines Partikels durch ein statisches Strömungsfeld zu sehen. Der Zeitschritt, der zur Berechnung verwendet wird, ist jeweils derselbe ($t_0 = t_1 = t_2$).

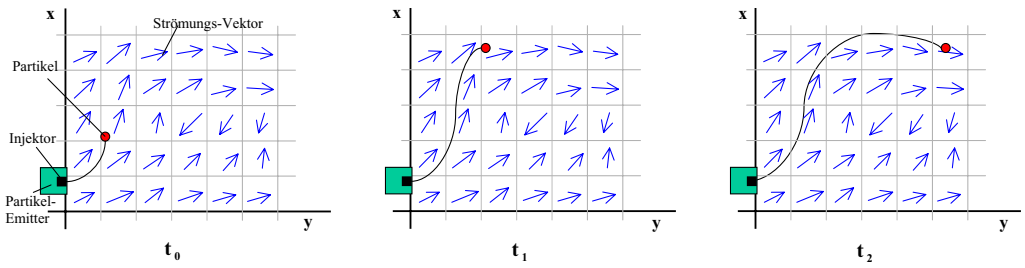



***Abbildung 7.12:*** *Darstellung einer Streamline in einem statischen Strömungsfeld*

**Timelines**

Bei Timelines werden eine gewisse Anzahl von Partikeln in ein nicht statisches, d.h. dynamisches, Feld emittiert. Der entscheidende Unterschied in der Darstellung zu den Streamlines liegt darin, dass jede weitere zu berechnende Position nun von dem darauffolgenden, also einem anderen, Vektorfeld herrührt, da sich pro Simulationszeitabschnitt das Strömungsfeld ständig verändert. Daraus folgt, dass immer nur die neueste Position am Ende der Strömungslinie ihre Koordinaten ändert. Die alten Linienteilstücke bleiben während der gesamten Simulation bestehen. Über die Zeit entsteht so eine immer länger werdende Linie.

Zur Verdeutlichung zeigt Abbildung 7.13 den Weg mehrerer gleichzeitig zum Zeipunkt $t_0$ emittierter Partikel durch ein sich veränderndes Strömungsfeld, wobei $t_0 \neq t_1 \neq t_2$.

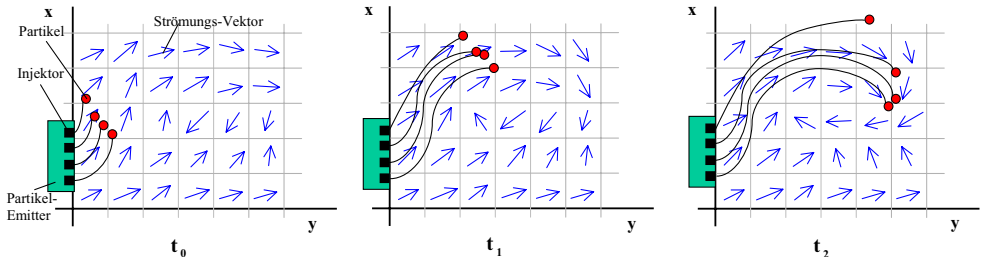



***Abbildung 7.13:*** *Darstellung von Timelines in einem veränderlichen Strömungsfeld*

**Streaklines**

Diese Art von Linienberechnung entspricht im Grunde genommen den Timelines bzgl. ihrer fortlaufenden Entstehung und dem zugrundeliegenden Strömungsfeld. Der Unterschied liegt in der Erzeugung der Streaklines: Bei jedem neuen Simulationsschritt wird eine neue Streakline an derselben Startposition erzeugt bzw. gezeichnet und während der fortlaufenden Simulation ständig aktualisiert. Die alte Linie wird zusätzlich weitergeführt. Pro Zeitschritt kommen also ständig neue Linien hinzu, während die vorhandenen immer länger werden. Dies hat den großen Vorteil einer effizienten und detailgetreuen Darstellung des Strömungsfeldes, geht aber auf Kosten der Performanz, da der Rechenaufwand pro Zeitschritt und Streakline durch die immer neu dazukommenden Linien ständig wächst.

In Abbildung 7.14 ist eine *Streakline* zu sehen. Hier ist $t_0 \neq t_1 \neq t_2$. Zur Verdeutlichung werden im Beispiel die Teilchenpositionen nicht durch Linien gekennzeichnet sondern durch Punkte dargestellt.

### 7.1.3.2 Positionsbestimmung / Integrationsalgorithmen

Die Positionsbestimmungen eines Teilchens in einem Strömungsfeld erfolgen mittels einer Integration der einfachen Differentialgleichung [Sti99]:

$$\frac{d\vec{x}}{dt} = \mathbf{u}(\vec{x}, t) \tag{7.6}$$

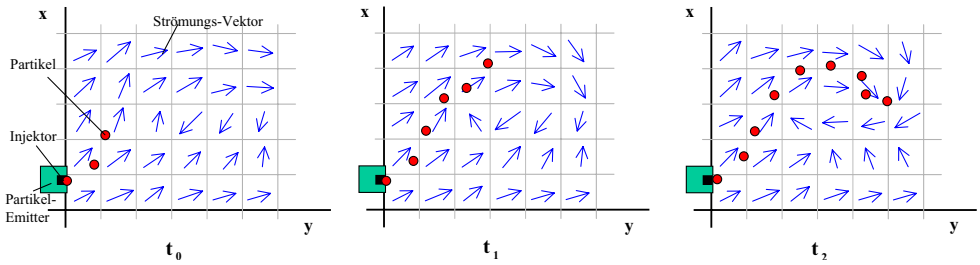



***Abbildung 7.14:*** *Darstellung einer Streakline in einem veränderlichen Strömungsfeld*

Hier steht $x$ für die Position des Teilchens, $t$ für die Zeit und $\mathbf{u}(\vec{x}, t)$ für die Geschwindigkeit zum Zeitpunkt $t$ an der Position $\vec{x}$ des Teilchens.

Der zur Lösung dieser Gleichung verwendete numerische Algorithmus wird als *Integrator* bezeichnet und ist angelehnt an das *Taylor-Verfahren* [BSM00]. Integratoren können in verschiedenen Varianten verwendet werden, die eine Balance zwischen Genauigkeit und Geschwindigkeit gestatten. Weitere Literatur zu diesem Thema ist in [Dar], [SvWHP94], [CP92] und [Kir02] zu finden.

Die im Folgenden aufgeführten Algorithmen wurden implementiert, um dem Benutzer verschiedene Geschwindigkeits- und Genauigkeitsgrade zu ermöglichen. Sie sind während der laufenden Visualisierung umschaltbar, wodurch eine gute Beobachtung der resultierenden Veränderung von Partikelpfaden bzw. der daraus entstehenden Linienzüge beobachtet werden kann.

Für die hier vorgestellten Algorithmen sei $\vec{x}$ die Partikelposition im Raum, $t$ der Zeitschritt, $\vec{u}\ (\vec{x}, t)$ die Partikelgeschwindigkeit an Position $\vec{x}$ zum Zeitschritt $t$, $h$ die Schrittweite, mit der die Berechnung durchgeführt werden soll und $k$ ein „Hilfsvektor“. $n$ bezeichnet den Iterationsschritt.

Die Schrittweite ist ein Faktor, der verwendet wird, um den aus der Funktion resultierenden Geschwindigkeitsvektor $\vec{u}$ zu beeinflussen. Somit kann die Fortbewegungsgeschwindigkeit der Partikel gesteuert werden. Die Verfahren können mit einer während des Programmablaufs festen oder adaptiven Schrittweite arbeiten.

### Euler Integrator

Die grundlegende Formel der Euler Methode lautet

$$\vec{x}_{n+1} \quad := \quad \vec{x}_n + h\ \vec{u}\ (\vec{x}_n, t_n). \tag{7.7}$$

Der Euler Integrator ist ein Algorithmus erster Ordnung und, seiner geringen Komplexität entsprechend, verhältnismäßig schnell. Er produziert jedoch Ergebnisse, die mitunter sehr ungenau sein können. Ferner kann es zur Entstehung künstlicher Merkmale

in der Visualisierung kommen [Ken]. Desweiteren bleiben charakteristische Stellen im Strömungsfeld möglicherweise unentdeckt. Diese Methode ist in erster Linie geeignet, um eine grobe „Abtastung“ des Strömungsfeldes vorzunehmen und damit vielversprechende Stellen schnell finden.

**Runge-Kutta Integrator zweiter Ordnung**

Die Formel für die Runge-Kutta Methode zweiter Ordnung (auch Heun Methode) lautet (siehe hierfür auch [Dar])

$$\begin{aligned} \vec{k}_1 &:= h\,\vec{u}\,(\vec{x}_n, t_n), \\ \vec{k}_2 &:= h\,\vec{u}\,(\vec{x}_n + \vec{k}_1, t_n + h), \\ \vec{x}_{n+1} &:= \vec{x}_n + \frac{1}{2}(\vec{k}_1 + \vec{k}_2). \end{aligned} \tag{7.8}$$

Dieser Integrator ist von hinreichender Genauigkeit, um charakteristische Merkmale des Strömungsfeldes zu entdecken und verbindet dies mit einem akzeptablen Rechenaufwand. Somit ist diese Methode geeignet für eine interaktive Analyse eines Strömungsfeldes.

**Runge-Kutta Integrator vierter Ordnung**

Um dem Benutzer die Möglichkeit zu geben, eine genauere Abtastung des Strömungsfeldes vorzunehmen, wird eine weitere Berechnungsmethode zur Verfügung gestellt. Diese kann zum Einsatz kommen, um Merkmale, die mit einem Algorithmus niedrigerer Ordnung entdeckt wurden, zu verifizieren bzw. künstlich erzeugte Merkmale möglichst auszuschließen. Die Formel dieser Methode beschreibt sich wie folgt

$$\begin{aligned} \vec{k}_1 &:= h\,\vec{u}\,(\vec{x}_n, t_n), \\ \vec{k}_2 &:= h\,\vec{u}\,(\vec{x}_n + \frac{1}{2}\,\vec{k}_1, t_n + \frac{1}{2}h), \\ \vec{k}_3 &:= h\,\vec{u}\,(\vec{x}_n + \frac{1}{2}\,\vec{k}_2, t_n + \frac{1}{2}h), \\ \vec{k}_4 &:= h\,\vec{u}\,(\vec{x}_n + \vec{k}_3, t_n + h), \\ \vec{x}_{n+1} &:= \vec{x}_n + \frac{1}{6}(\vec{k}_1 + 2\,\vec{k}_2 + 2\,\vec{k}_3 + \vec{k}_4). \end{aligned} \tag{7.9}$$

Neben den hier vorgestellten sind noch weitere Integrationsalgorithmen für die Anwendung auf dieses Problem denkbar (z.B. Verlet [ver01]). Im Rahmen dieser Arbeit wurden exemplarisch die hier vorgestellten Algorithmen realisiert.

Diese Methode wurde weiterhin in einer Abwandlung mit adaptiver Schrittweite implementiert. In diesem Verfahren wird nach jeder Berechnung ein Fehlerwert ermittelt. Ist

dieser größer als ein vorher festgelegter maximaler Fehler, so wird die Berechnung mit verringerter Schrittweite nochmals durchgeführt. Dies geschieht so lange, bis der Fehlerwert unter dem Schwellenwert liegt. Die Berechnung sämtlicher Folgepartikel dieses Zeitschrittes beginnt mit der verringerten Schrittweite. Damit wird die Schrittweite an die Teilchen gebunden, die sich am unruhigsten verhalten.

#### 7.1.3.3 Emitter

In der Elektrotechnik bezeichnet ein sogenannter Emitter den Teil eines Transistors, der die Elektronen liefert. In unserem Fall ist es ein Objekt, das vom Anwender in der Szene bewegt und manipuliert (Skalierung, Rotation) werden kann. Auf der Oberfläche dieses Emitters werden zu Beginn der Simulation Injektionspunkte erzeugt, die ihrerseits als Startpunkte für die Partikel dienen, welche dann die Strömungslinien erzeugen, indem sie mit fortlaufender Zeit durch das Strömungsfeld bewegt werden. Dem Partikelemitter kommt die Aufgabe der Festlegung des Ursprungs der zu erzeugenden Partikel zu.

Grundsätzlich ist als Partikelemitter fast jede Form denkbar. Von einer Linie über eine Röhre bis hin zu Flächen oder Quadern kann jedes Objekt in Frage kommen. Die Anforderungen an dieses Objekt richten sich nach dem Anwendungsgebiet. Es ist jedoch immer wünschenswert, einen ausreichend flexiblen Emitter zur Verfügung zu haben, um den Partikel-Ursprung möglichst wunschgemäß festlegen zu können.

Für die Realisierung des Emitters der Visualisierungsmethode wurde ein Quader ausgewählt, von dem eine Fläche als Partikelemmissionsfläche verwendet wird (siehe Abbildung 7.15).

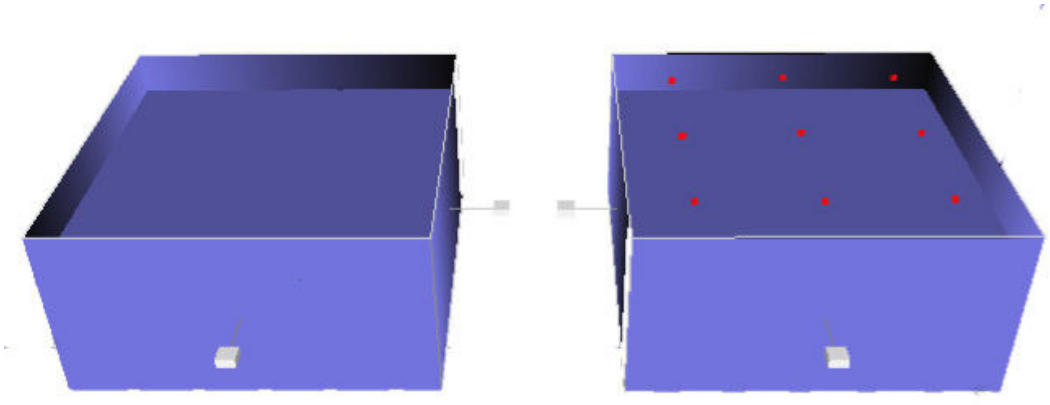


***Abbildung 7.15:*** *Partikelemitter ohne und mit Injektoren*

Zur Interaktion mit dem Emitter wurde ein sogenannter *Manipulator Knoten* aus dem Szenegraph verwendet. Mit seiner Hilfe ist es möglich zum einen den Emitter in seiner Größe zu verändern, als auch ihn zu drehen. Der Emitter kann innerhalb der Probe hin und her bewegt werden (Abbildung 7.16).

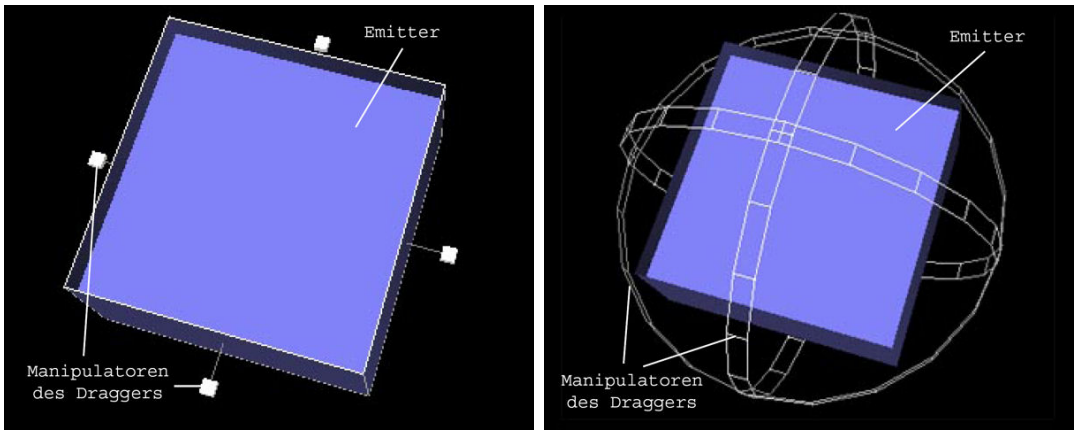



***Abbildung 7.16:*** *Resize und Rotation Manipulator des Emitters*

#### 7.1.3.4 Injektoren

Der Begriff Injektor bezeichnet oftmals in der Elektrotechnik eine zusätzliche Elektrode zum Injizieren von Ladungsträgern bei speziellen Halbleiterbauelementen. Im Falle der Visualisierungsmethode Streamlines wiederum repräsentiert ein Injektor den genauen Startpunkt einer Strömungslinie und hat daher eindeutige Koordinaten in dem vorliegenden dreidimensionalen Strömungsraum.

Injektoren sind mit Hilfe des Partikelemitters vom Anwender steuerbar und können damit beliebig positioniert werden. Sie sind demzufolge mit verantwortlich für den Weg, den die Partikel durch das Strömungsfeld nehmen. Ungünstig positionierte Injektoren können dazu führen, das wichtige Details im Strömungsfeld unentdeckt bleiben oder nur schlecht dargestellt werden. Daher ist es bei variierenden Strömungsfeldern wichtig, dem Anwender sehr flexible Möglichkeiten zur Injektorplatzierung zu gestatten, um ein akkurates Darstellungsergebnis erzielen zu können.

Die Anordnung und Anzahl der Injektionspunkte auf dem Emitter wird hier als Partikel-Emissionsart bezeichnet. Dem Vorsatz der Flexibilität folgend sind mehrere Möglichkeiten der Partikelemission implementiert und werden vom Anwender über die Benutzeroberfläche ausgewählt. Verschiedene vorstellbare Anwendungsszenarien lassen es sinnvoll erscheinen, die Injektorenanordnung weitgehend kontrollieren zu können. So kann es sinnvoll sein, die Bahn eines einzelnen Partikels („Single“) durch ein Srömungsfeld zu verfolgen, um dessen Bewegung exakt nachvollziehen zu können. Eine Reihe von Partikeln („Line“) kann gleich einem Teppich in die Szene emittiert werden und läßt durch entstehende Wellenbewegungen Rückschlüsse auf Turbulenzen im Strömungsfeld zu. Ein Injektorfeld („Field“) kann verwendet werden, um einen Raum großflächig mit Partikeln zu bestreuen. Alle Emissionsarten, mit Ausnahme von „Single“, lassen es zu, die Anzahl erzeugter Injektoren in gewissen Grenzen zu verändern. Je nach Anordnung der Injektoren auf dem Emitter lassen sich verschiedene Partikelemissionsarten unterscheiden:

**Single**

Ein *Single-Injektor* platziert, wie in Abbildung 7.17 zu sehen ist, einen einzigen Injektor in der Mitte der Emitteroberfläche. Diese Art der Emission bietet die Möglichkeit, die Bahn eines Partikels sehr genau zu verfolgen, da er nicht in einer Masse vieler Partikel „verschwindet“. Außerdem kann man sich den zurückgelegten Pfad des Partikels anzeigen lassen und nötigenfalls auch festhalten, nachdem er das Strömungsfeld bereits verlassen hat.

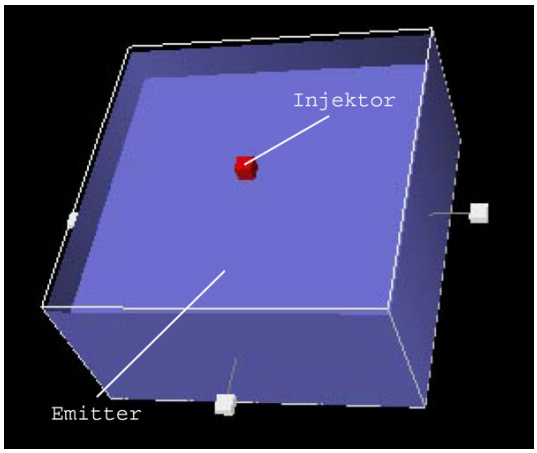



***Abbildung 7.17:*** *Ein mit „Single-Injektor“ erzeugter Injektor auf einem Emitter*

**Line**

Der *Line-Injektor* erzeugt eine Reihe von Partikeln auf dem Emitter. Über die Injektoranzahl läßt sich steuern, wie viele Injektoren erzeugt werden und in welchen Abständen diese platziert werden. Bei genügend hoher Anzahl entsteht somit eine geschlossene Linie. Weiterhin kann der Line-Injektor auf dem Emitter verschoben und in seiner Ausrichtung verändert werden (Drehung um 90°). Abbildung 7.18 zeigt eine solche Injektorreihe. Auf der linken Seite ist die Ausgangspositionierung eines Line-Injektors zu sehen. Auf der rechten Seite ist er um 90° gedreht und leicht zum Rand hin verschoben. Bei der fortgesetzten Partikelemission (Streakline) aus einer Injektorreihe erzeugen die Partikel eine Art „Teppicheffekt“. Dieser ist geeignet, um Wirbel und Turbulenzen im Strömungsfeld zu visualisieren, denn an solchen Stellen bildet der Teppich Wellen, die leicht erkennbar sind.

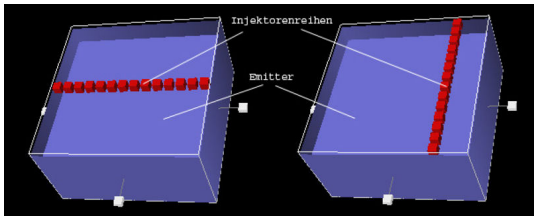



***Abbildung 7.18:*** *Zwei unterschiedliche mit „Line-Injektor“ erzeugte Injektorenreihen auf einem Emitter*

**Field**

*Field-Injektoren* unterteilen sich nochmals in *Regular Field-* und *Random Field-Injektoren*. Beiden ist gemeinsam, dass sie die komplette Emitteroberfläche, soweit es die gewählte Injektoranzahl zuläßt, mit Injektoren bedecken.

Regular Field-Injektoren platzieren Injektoren entsprechend der Länge und Breite des Emitters in regelmäßigen Abständen in Form eines Gitternetzes auf dem Emitter. Ausgehend von der geforderten Injektoranzahl werden die benötigten Reihen und Spalten berechnet, die eine möglichst gleichmäßige Verteilung gewährleisten. So werden beispielsweise 16 Injektoren auf einem exakt quadratischen Emitter auf einem 4x4-Gitter angeordnet. Auf einem rechteckigen Emitter mit einem Länge-Breite-Verhältnis von 2:1 würden die Injektoren in zwei Reihen und acht Spalten unterteilt. Sollte die Injektoranzahl eine genau reihen- und spaltenweise Aufteilung nicht zulassen, so wird automatisch die nächstmögliche niedrigere Anzahl gewählt. Diese Art der Injektoraufteilung wird in Abbildung 7.19 verdeutlicht.

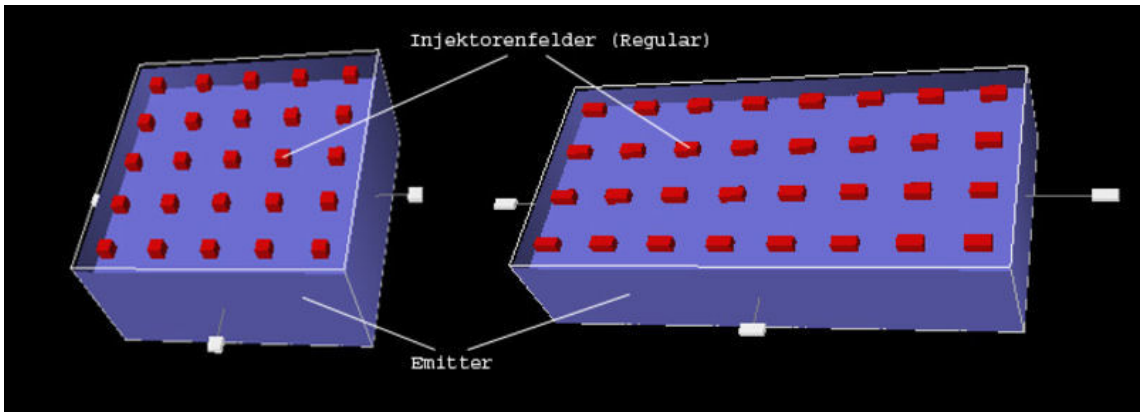



***Abbildung 7.19:*** *Zwei unterschiedliche mit „Regular Field-Injektor“ erzeugte Injektorenfelder auf einem Emitter*

Random Field-Injektoren erzeugen Injektoren in der gewünschten Anzahl zufällig verteilt auf der Fläche des Emitters. Dies wird in Abbildung 7.20 veranschaulicht. Eine solche Art der Verteilung von Injektoren kann sinnvoll sein, um der Gleichmäßigkeit eines erzeugten Partikelstroms und entgegenzuwirken, welche die Strömungslinien mitunter konstruiert bzw. künstlich aussehen lassen kann.

#### 7.1.3.5 Beleuchtete Strömungslinien

Die Verwendung von Beleuchtung ist ein wichtiger Aspekt bei der Darstellung von dreidimensionalen Graphikanimationen und Bildern. Unter Einsatz von virtuellen Lichtquellen ermöglicht man dem Betrachter, tiefenrelevante Informationen und ein besseres räumliches Vorstellungsvermögen der dargestellten Strömungslinien zu erhalten. Im Grunde genommen stellt eine Beleuchtung von Linien auch eine Art Einfärbung dar. Beleuchtung nutzt im Gegensatz zu einfachen Einfärbungskonzepten zusätzlich die Möglichkeit aus, durch Schatten, Glanzlichter und die dazwischenliegenden Graustufen die Darstellungen realistischer und

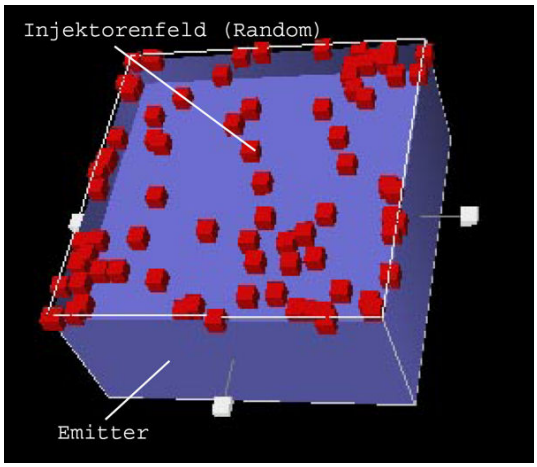



***Abbildung 7.20:*** *Ein mit „Random Field-Injektor“ erzeugtes Injektorenfeld auf einem Emitter*

interpretierbarer zu machen. Hierdurch wird es z.B. ermöglicht, Tiefen, Krümmungen und starke Wölbungen der Strömungslinien innerhalb des Strömungsfeldes hervorzuheben. Man darf nicht vergessen, dass der Bildschirm nur eine zweidimensionale Fläche repräsentiert, auf der man ohne Lichteffekte und geschickte Einfärbungen keinen räumlichen Eindruck dessen bekommen kann, was man sieht. Durch das Verfahren, Linien zu beleuchten, ist es hingegen möglich, durch entstehende Glanzlichter an bestimmten Stellen den Eindruck eines räumliches Bildes zu erhalten, um z.B. kleine Strudel und Biegungen der Linien in der Strömung leichter entdecken zu können.

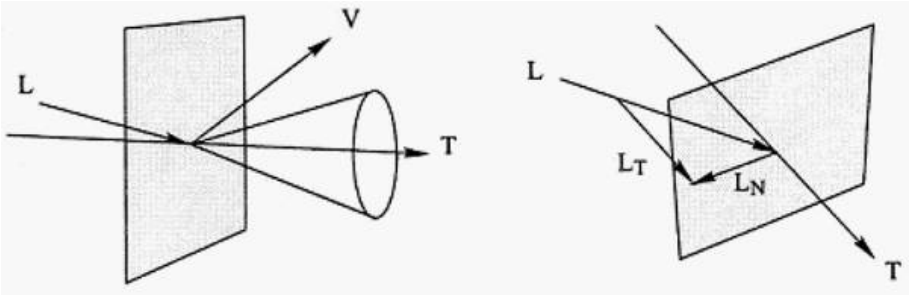



***Abbildung 7.21:*** *Lichteinfall auf einer Linie im 3D Raum: Der Normalenvektor ist nicht eindeutig bestimmt. Vielmehr exisitert eine Normalenebene [ZSH].*

Für die notwendigen Berechnungen der Lichtintensität wird das vorgestellte Phong-Beleuchtungsmodell (siehe Kapitel 3.6.1) verwendet. Nun steht man aber vor folgendem Problem: Der Normalenvektor $\vec{N}$ und damit auch der Reflektionsvektor $\vec{R}$ (siehe Abbildung 7.21) können im Falle von Linien nicht eindeutig bestimmt werden, da eine Linie im Gegensatz zu einer Fläche unendlich viele Normalenvektoren, angeordnet auf einer Ebene, besitzt. An dieser Stelle kann man sich eines einfachen Tricks bedienen. Demnach verändert man die mathematischen Abhängigkeiten unter Zuhilfenahme eines eindeutigen Tangentenvektors $\vec{T}$, der zum betreffenden Liniensegment gehört, für das man die Lichtberechnung durchführt.

Anschließend wählt man den Normalenvektor aus, der sich in der gleichen Ebene wie $\vec{L}$ und $\vec{T}$ befindet.

Nach Anwendung einfacher Umformungen der Formel aus Kapitel 3.6.1 der Skalarprodukte $\vec{L} * \vec{N}$ und $\vec{V} * \vec{R}$ unter Berücksichtigung geometrischer Gesetzmäßigkeiten und Normierung der Vektoren $\vec{L}$, $\vec{N}$, $\vec{V}$ und $\vec{R}$ ergeben sich folgende Gleichungen:

$$\vec{L} * \vec{N} = \left|\vec{L_N}\right| = \sqrt{1 - (\vec{L} * \vec{T})^2} \tag{7.10}$$

$$\vec{V} * \vec{R} = (\vec{L} * \vec{T})(\vec{V} * \vec{T}) - \sqrt{1 - (\vec{L} * \vec{T})^2} * \sqrt{1 - (\vec{V} * \vec{T})^2} \tag{7.11}$$

Eingesetzt in die Formel zum Phong Beleuchtungsmodell ergibt sich die Lichtintensität $I$ an einem bestimmten Punkt somit nach folgender Gleichung:

$$I = k_a + k_d(\sqrt{1 - (\vec{L} * \vec{T})^2}) + k_s \left( (\vec{L} * \vec{T})(\vec{V} * \vec{T} - \sqrt{1 - (\vec{L} * \vec{T})^2} * \sqrt{1 - (\vec{V} * \vec{T})^2} \right)^m \tag{7.12}$$

Nach dieser Umformung hängt die komplette Berechnung der Lichtintensität an einem Punkt nur noch von drei Komponenten ab, die in Form von Vektoren vorliegen. Sie sind im Einzelnen: Erstens die Position des Lichtes $\vec{L}$, zweitens die Position des Betrachters $\vec{V}$ und drittens ein Tangentenvektor $\vec{T}$ des entsprechenden Liniensegmentes. Bei Veränderung einer dieser Faktoren ändert sich auch die Beleuchtung und damit die Lage der Glanzlichter auf den Linien. Um nun die permanente Berechnung eines geeigneten Normalenvektors zum jeweiligen Linienstück zu umgehen, wird im Folgenden ein Ausweg vorgestellt, um die CPU zu ent- und stattdessen den Graphikprozessor bestmöglichst auszulasten. Gleichzeitig wird das Verfahren des Multitexturing eingesetzt, um eine größtmögliche Flexibilität, Erweiterbarkeit und Gesamtperformanz bzgl. des Programms und des Ablaufs der Simulation zu erreichen. Das Konzept und die prinzipielle Realisierung von beleuchteten Strömungslinien ist unter [IllumLines1] nachzulesen. Die Steuerung der Werte für die zur Lichtberechnung benötigten drei Konstanten, den diffusen (ausbreitende, diffundierende), spekularen (glänzende, reflektierende) und shininess (Glanz) Anteil des Lichtes, soll über das zugehörige Benutzerinterface möglich sein. Den ambienten (umgebenden) Anteil stellt die Grundfarbe dar.

Um eine hardwarebeschleunigte Beleuchtung durch Texturierung der Linien realisieren zu können, ist eine weitere Umformung der oben vorgestellten Formeln nötig, um die Produkte $\vec{L} * \vec{T}$ und $\vec{V} * \vec{T}$ zu ersetzen. Das Ziel ist es, nur durch Verwendung von einer zweidimensionalen Textur, die entsprechenden Lichtintensitäten anzusteuern.

Nun wird ein Texturvektor $t_0$ bestimmt, der dem Tangentenvektor eines kompletten Liniensegments entspricht. Anschließend werden beim Zugriff auf die Texturkoordinaten diese mit einer 4x4 Transformationsmatrix $M$ (siehe nachfolgende Formel) multipliziert. Diese Vorgehensweise unter Benutzung der Transformationsmatrix $M$ ist unerlässlich und sehr wichtig, um eine hardwarebeschleunigte Umsetzung der beleuchteten Linien realisieren zu können.

$$M = \frac{1}{2}\begin{bmatrix} L_1 & V_1 & 0 & 0 \\ L_2 & V_2 & 0 & 0 \\ L_3 & V_3 & 0 & 0 \\ 1 & 1 & 0 & 2 \end{bmatrix} \tag{7.13}$$

$L_1$, $L_2$, $L_3$, $V_1$, $V_2$, $V_3$ stellen die einzelnen Raumkoordinaten in x-, y- und z-Richtung des Licht- bzw. Betrachtervektors dar. Durch die durchgeführte Multiplikation ergibt sich aus $t = t_0 * M$ nach Umformung das folgende:

$$t_1 = \frac{1}{2}(L * T) + 1) \tag{7.14}$$

$$t_2 = \frac{1}{2}(V * T) + 1) \tag{7.15}$$

$t_1$ und $t_2$ sind die resultierenden Texturkoordinaten in x- und y-Richtung, mit denen man letztendlich auf die zuvor berechnete zweidimensionale Textur im späteren Verlauf zugreift. Sie nehmen nur Werte zwischen 0 und 1 an (vgl. Abbildung 7.22: $\vec{V} * \vec{T}$ stehen für den spekularen und $\vec{L} * \vec{T}$ für den diffusen Anteil bzgl. der Lichtintensität).

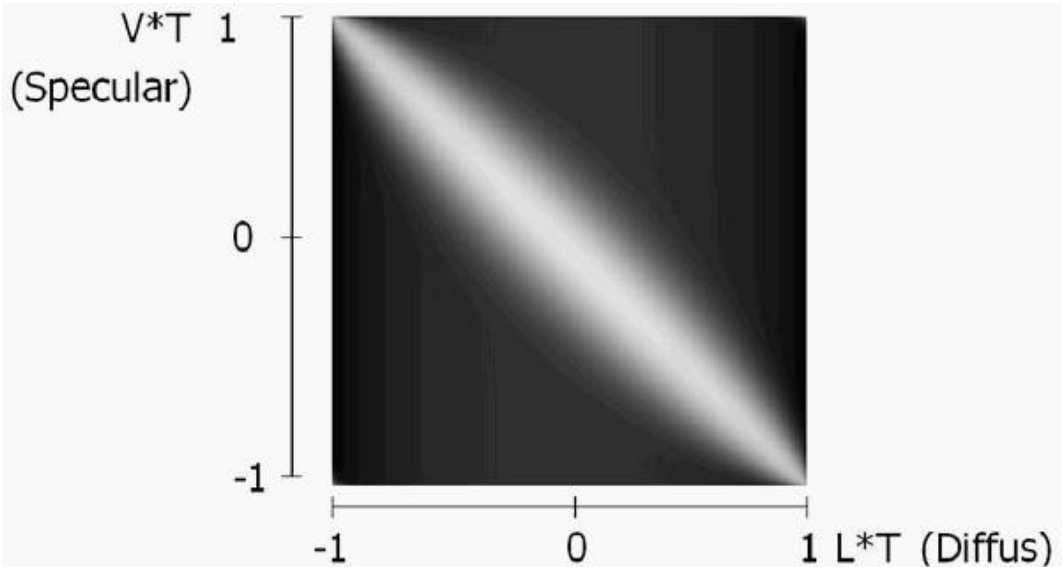



***Abbildung 7.22:*** *Nach dem Phong-Beleuchtungsmodell berechnete 2D Beispieltextur für das Streamline Shading.*

Bei Veränderung der Konstanten $k_a$, $k_d$ und $k_s$, die die Größen des ambienten, diffusen und spekularen Lichtanteils beeinflussen, wird eine neue zweidimensionale Textur mit 256 Wertepaaren generiert. Dann wird, falls sich die Betrachter- oder Lichtposition geändert hat, die 4x4 Matrix $M$ aktualisiert. Anschließend wird der benötigte Tangentenvektor berechnet und mit $M$ multipliziert. Dann greift man auf die entsprechenden Koordinaten der 2D Shading

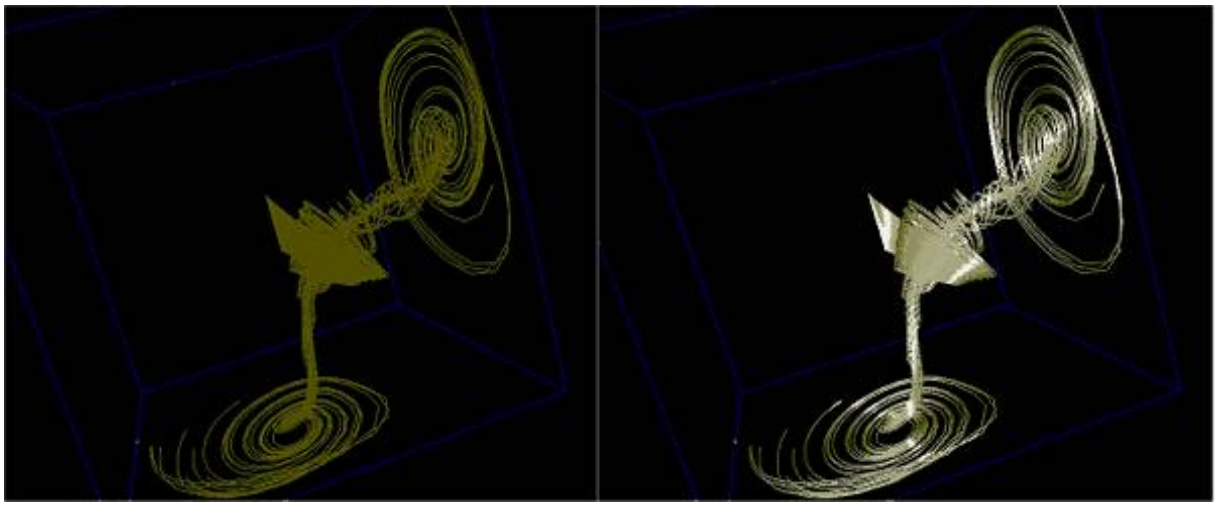


***Abbildung 7.23:*** *35 einfarbige Streamlines: links ohne Beleuchtung, rechts mit Beleuchtung ($k_d = 0.5$, $k_s = 0.8$, $n = 80$).*

Textur zu, die die verschiedenen Lichtintensitäten, abhängig von den Vektoren $\vec{L}$, $\vec{V}$ und $\vec{T}$, repräsentiert. Abbildung 7.23 zeigt beleuchtete Strömungslinien.

Nachdem nun festgelegt ist, wie die Lichtintensität auf ein Liniensegment wirkt, fehlt jetzt noch der ambiente Teil, der von der Ursprungsfarbe der Linie ohne Beleuchtung dargestellt wird (dies könnte also z.B. die Farbe der einstellbaren Farbpalette sein), und der mit dem berechneten Grauwert für das Glanzlicht gemischt werden soll. Dieses „Mischen“ der zwei Farbanteile geschieht im realisierten Beispiel unter Zuhilfenahme von Multitexturing. Um die Zielfarbe $Z$ zu erhalten, wird folgende Formel angewendet: $Z$ = (Ursprungsfarbe * (1 - Lichtintensität)) + Lichtintensität. Die Lichtintensität und Ursprungsfarbe können Werte zwischen 0 und 1 annehmen. Dies würde einer 0%igen und 100%igen Gewichtung bzw. einem 0%igen und 100%igen Anteil des Ursprungswertes entsprechen. Angenommen die Lichtintensität wäre 0, d.h. kein Licht wäre vorhanden, und die Ursprungsfarbe wäre weiß (dies entspräche im RGB-Modell dem höchsten Wert, also 1), dann würde $Z$ der ursprünglich eingestellten Farbeinstellung entsprechen. Ist hingegen die Lichtintensität am höchsten, also 1, dann wäre die Zielfarbe weiß, unabhängig von der gewählten Ursprungsfarbe. Werte im Bereich von 0 und 1 ergeben entsprechend Zwischenwerte bzgl. der Farbmischung (eine Ursprungsfarbe Blau mit geringer Beleuchtung ergäbe z.B. ein helles Blau). Jeder Texturunit des Multitexturings kann nun eine Textur und eine Operation zugewiesen werden. Für die obige Formel werden deshalb drei Units benutzt. Die erste enthält die Ursprungsfarbe und überschreibt eventuell vorhandene alte Farbkomponenten. Der zweiten wird der invertierte Wert der berechneten Lichttextur zugewiesen (1 - Lichtintensität) und mit der Ursprungsfarbe gemischt (in Form einer Multiplikation (Ursprungsfarbe * (1 - Lichtintensität))). Die dritte Unit besitzt diesmal den Originalwert der Lichtintensität (Lichtintensität) und wird nun lediglich noch zum bisherigen Ergebnis addiert. Das Ergebnis ist die Zielfarbe $Z$.

#### 7.1.3.6 Parallelisierung

Der Kern der Visualisierungsmethode besteht aus drei großen Teilen:

- Klasse *Particles* (Steuerklasse)
- Klasse *ParticleEngine* (Verwaltung der Partikel und des Emitters)
- Klasse *StreamLinesWorker* (Streamlines Aktualisierung)

Die übergeordnete Verwaltung und Ablaufsteuerung wird von der Klasse `Particles` übernommen. Die Klasse `ParticleEngine` verwaltet sämtliche Partikel und den Emitter. Die Klasse `StreamLinesWorker` ist zuständig für die Berechnung der Partikelpfade. Da dies den größten Rechenaufwand erfordert, bietet sich diese Klasse zur Parallelisierung an. Daher wird sie als Threadklasse realisiert, d.h. von der in Kapitel 6.6.2 vorgestellten `WorkerThread` Klasse abgeleitet. Zur Laufzeit sind mehrere Instanzen der Workerklasse aktiv.

**Klasse Streamlines Vismodule**

Die Klasse `StreamlinesVismodule` sorgt für die Anmeldung des Partikelvisualisierungsmoduls beim System und initialisiert die Erstellung der Benutzeroberfläche. Außerdem verwaltet sie sämtliche Daten, die das Erscheinungsbild und das Verhalten der Linien steuern. Dazu gehören zum Beispiel die Farbe und die Injektionsart. Die Steuerklasse übernimmt weiterhin die Funktion einer Schnittstelle zwischen der Benutzeroberfläche und der Particle Engine. Als Ausgangspunkt eines jeden Berechnungsdurchgangs löst sie für jeden Zeitschritt die notwendigen Partikelberechnungen und eventuell neue Initialisierungen aus. Sie übernimmt außerdem die Koordination der aktiven Threads. Die Überwachung der Zeitschritte und die Benachrichtigung der Klasse `Particles` darüber, dass ein neuer Zeitschritt zur Berechnung vorliegt, übernimmt der UpdateManager des Systems.

**Klasse ParticleEngine & Klasse Particle**

Der Kern des Visualisierungsmoduls ist die Klasse `ParticleEngine`, die drei wichtige Funktionen erfüllt. Ihre erste Aufgabe ist es, Partikel zu erzeugen, zu initialisieren und zu verwalten. Dies geschieht mit Hilfe einer weiteren Klasse namens `Particle`, welche ein einzelnes Partikelobjekt und die ihm zugehörenden Eigenschaften (wie zum Beispiel Position, Lebensspanne und Status) beschreibt. Für die graphische Darstellung werden die berechneten Positionen zu einer Linie verbunden. Die zweite Aufgabe der Klasse `ParticleEngine` ist die Erzeugung, Verwaltung und Kontrolle des Partikelemitters. Hier wird insbesondere darauf geachtet, dass die Abmessungen des Emitters keine unerwünschten Werte (zum Beispiel durch eine Skalierung mit 0) annehmen und er den Probenbereich nicht verläßt.

**Klasse StreamLines Worker**

Die zyklische Aktualisierung der Partikel findet in der Klasse `StreamLinesWorker` statt. Hier werden die neuen Positionen der Partikel entsprechend des gewählten Integrationsalgorithmus berechnet. Zusätzlich erfolgen hier sämtliche Validierungsprozesse

für die Partikel. Dazu gehören die Überprüfung der Lebenszeit, die Positionskontrolle (außerhalb des Probenbereichs befindliche Partikel werden deaktiviert) und verschiedene andere Mechanismen, die der Zustandsüberwachung der Partikel bzw. Strömungslinien dienen. Dies ist der leistungs- und zeitaufwendigste Teil der Visualisierung, der sich daher zur Parallelisierung anbietet. Bei Vorhandensein von mehreren Prozessoren ist es somit möglich, die Partikelaktualisierung auf mehrere Workerinstanzen zu verteilen und parallel abarbeiten zu lassen.

Sämtliche Partikel werden in einer einzigen Liste verwaltet. Bei jedem Update wird die gesamte Partikelliste auf die einzelnen Threads verteilt, und jedem Thread wird wiederum ein Startindex und eine Anzahl zu bearbeitender Partikel übergeben.

Jeder Partikel wird vom zuständigen Thread auf seinen Zustand überprüft, bevor er bearbeitet wird. Partikel, die deaktiviert werden, behalten ihren Platz in der Liste bei. Die Schwierigkeit hierbei ist es, die Threads möglichst gleichmäßig auszulasten. Da jedem Thread ein Teil der gesamten Partikelliste übergeben wird, ist es grundsätzlich möglich, dass ein Thread einen Listenteil voller aktiver Partikel erhält, während ein anderer nur inaktive Partikel in seinem Listenteil vorfindet. Die Lastverteilung unter den Threads wäre dementsprechend asymmetrisch. Um dies zu vermeiden, wird ein Algorithmus benötigt, der zum Initialisierungszeitpunkt alle zu aktivierenden Partikel möglichst gleichmäßig über die Liste verteilt. Abbildung 7.24 zeigt den Vorgang einer solchen Initialisierung.

Der Vorteil bei dieser Methode liegt in der Zeitersparnis durch die nun nicht mehr notwendige Sortierung. Desweiteren bietet er gute Voraussetzungen zur Parallelisierung. Der offensichtliche Nachteil ist die notwendige Bearbeitung der kompletten Liste zu jedem Update- und Renderzeitpunkt. Die Anzahl tatsächlich aktiver Partikel ist hierbei irrelevant. So wird, selbst im Falle eines einzigen aktiven Partikels und einer Listengröße von 10000 Partikeln, diese einmal vollständig durchlaufen. Um den möglichen Effizienzverlust klein zu halten, werden inaktive Partikel sofort nach dem Erkennen dieses Zustandes übergangen.

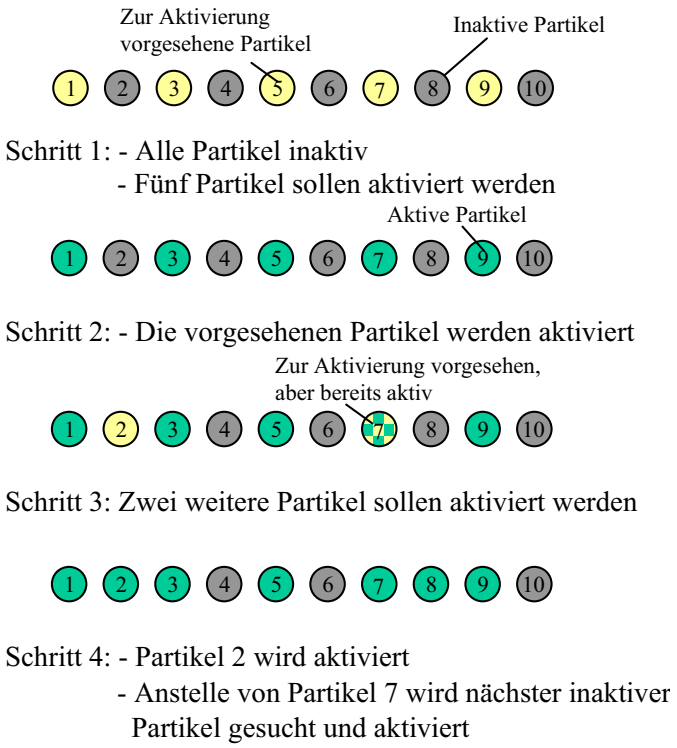



***Abbildung 7.24:*** *„Intelligente" Initialisierung von Partikeln unter Verwendung einer einzigen Partikelliste. In einer Liste aus 10 inaktiven Partikeln sollen 5 Partikel aktiviert werden. Nach der Berechnung der günstigsten Verteilung werden die Partikel 1, 3, 5, 7 und 9 aktiviert. Im folgenden Schritt sollen zwei weitere Partikel aktiviert werden. Zur Aktivierung werden die Partikel 2 und 7 ausgewählt. Da Partikel 7 aber bereits aktiv ist, wird der nächste inaktive Partikel gesucht und an dessen Stelle aktiviert.*

#### 7.1.3.7 Panel

Das zur Visualisierungsmethode passende Panel wird in Abbildung 7.25 gezeigt. Das Panel bildet eine Vielzahl von Einstellparametern ab, von denen hier nur die wichtigsten erläutert werden.

**Streamlines Methods**
Hier wird ausgewählt, welche Art Linien berechnet werden sollen. Zur Wahl stehen Streamlines (= fester Zeitschritt), Streaklines und Timelines.

**Injection Styles**
Hier kann die Art des Injektors festgelegt werden (z.B. Line Injektor oder Field Injektor).

**Integrator**
In diesem Bereich läßt sich auswählen, welcher numerische Integrator zur Berechnung der Linienbahnen verwendet werden soll. Je nach Typ entstehen hier unterschiedliche Linien, die sich in Berechnungsgeschwindigkeit und Genauigkeit unterscheiden.

**Colorchooser**
Hier kann man die Linie entlang ihrer Länge mit einem einstellbaren Farbverlauf belegen.

**Threads**
Neben der Anzahl der Threads, die zur Berechnung der Linien verwendet werden sollen, besteht hier auch die Möglichkeit, die Linien anhand des Threads, indem sie entstanden sind, einzufärben.

**Illumination**
In diesem Teil wird die größe und Erscheinungsform des Glanzlichtes auf den Linien gesteuert.

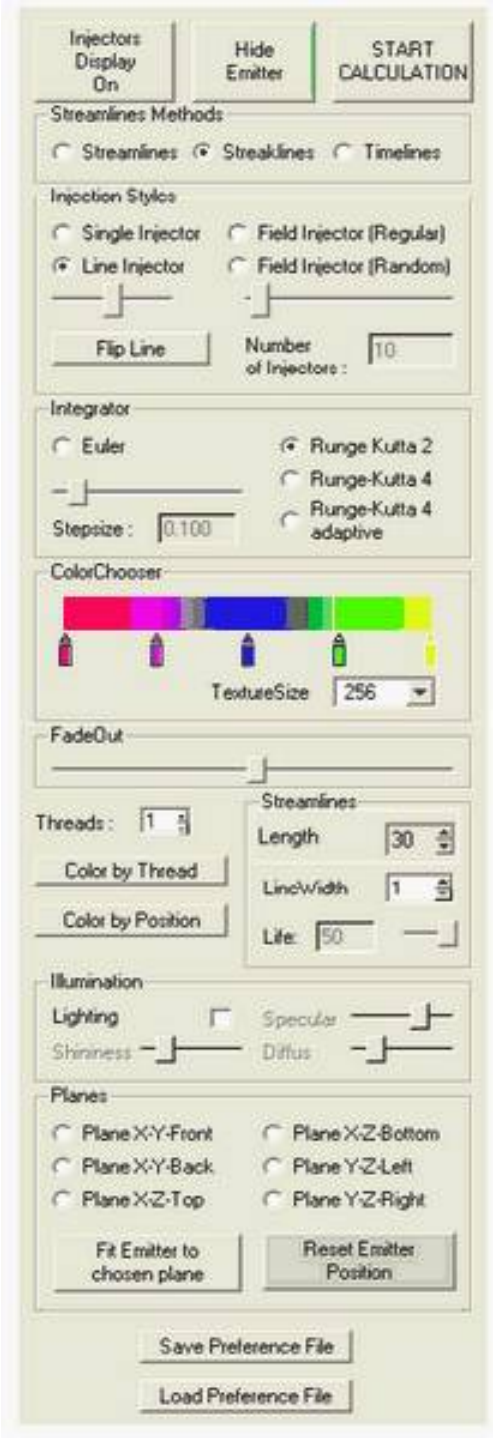



***Abbildung 7.25:*** *Panel zur Streamline Visualisierungsmethode.*

#### 7.1.3.8 Klassenstrukur

Ähnlich der bereits vorgestellte LIC Visualisierungsmethode (Kapitel 7.1.2), wird auch die Strömungslinien Darstellung in das Visualisierrungssystem integriert. Im direkten Vergleich kommt hier allerdings noch eine wesentliche Komponente hinzu: Die Partikel Engine Klasse mit der zugehörigen Partikelklasse (Abbildung 7.26).

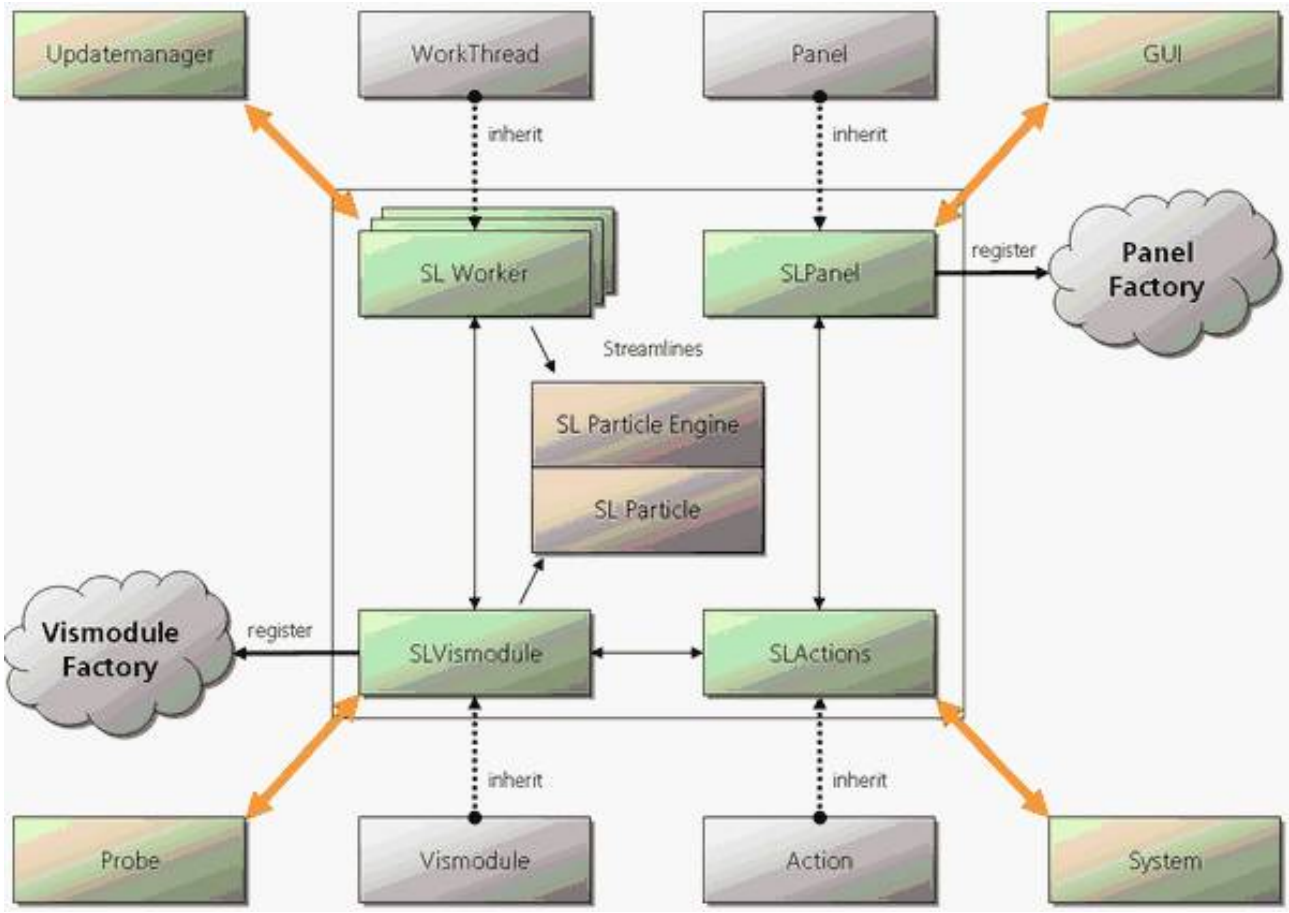
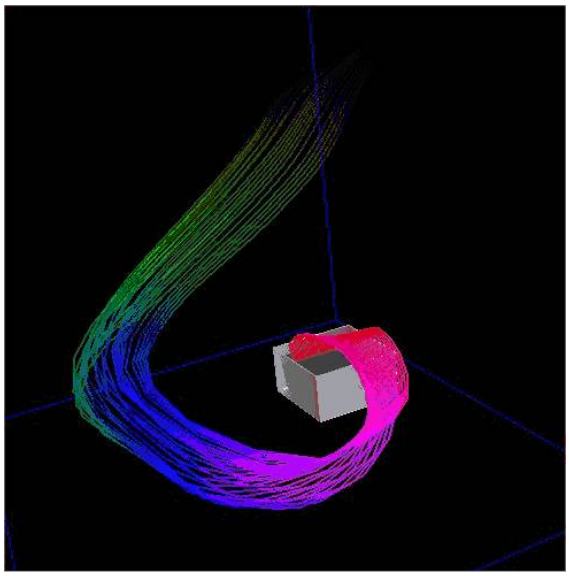


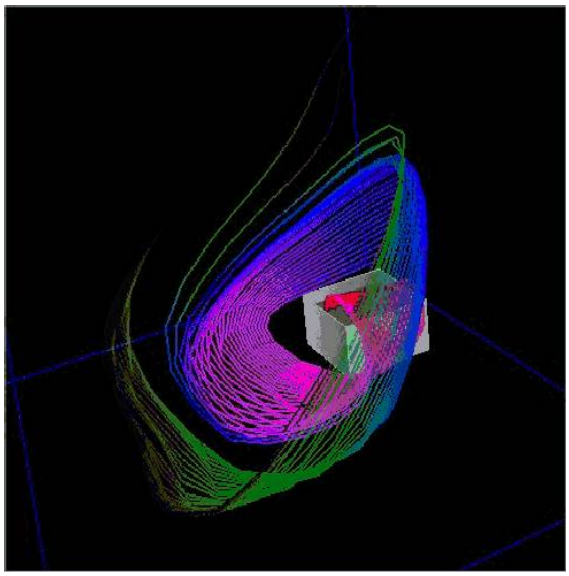

***Abbildung 7.26:*** *Die Klassen, die für die Implementierung des Strömungslinien Verfahrens verwendet werden.*

#### 7.1.3.9 Ergebnisbilder

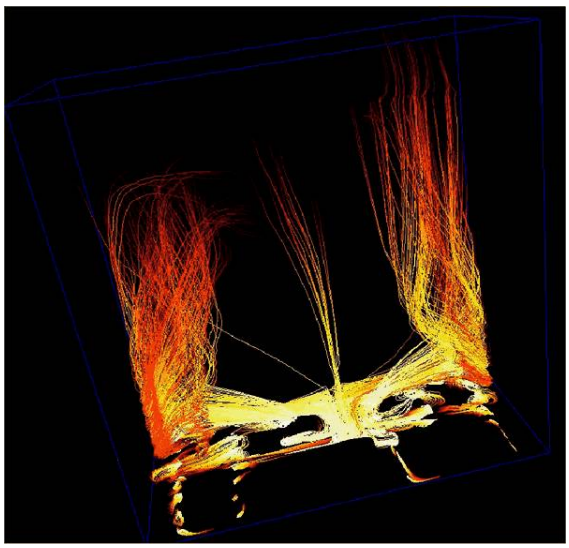
Hier einige Ergebnisbilder der Streamline Visualierungsmethode (Abbildung 7.27):



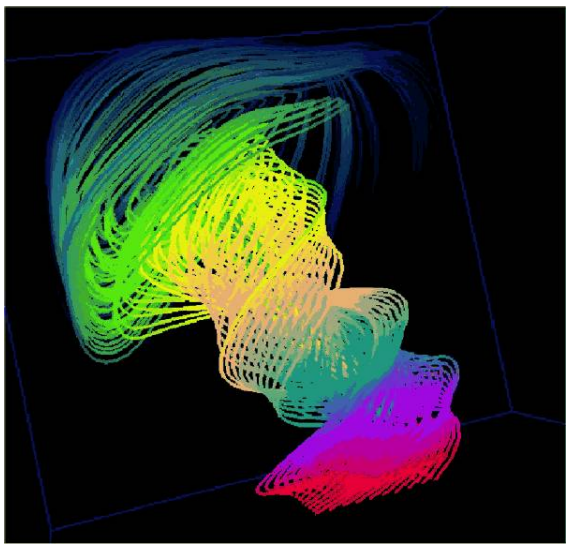
Positionsberechnung durch Eulerverfahren



Positionsberechnung durch Runge-Kutta Verfahren 4. Grades



Darstellung der Simulation eines Raumbrandes mit 60 Streaklines



Wirbelturbulenz: 7 Verschiedene Farben wurden hierfür im Farbverlaufbalken eingestellt.

***Abbildung 7.27:*** *Verschiedene Parametereinstellungen und deren Resultate im Strömungslinien-Verfahren.*

### 7.1.4 Partikel Rendering

Der Aufbau der Partikelvisualisierungsmethode unterscheidet sich nur an einigen Stellen von der Streamlines Methode. Auch bei ihr steht das Visualisierungssystem vor der Aufgabe, die Bahnen mehrerer massenloser Partikel, die von Injektoren losgeschickt werden, mit Hilfe verschiedener „Integratoren“ durch das Strömungsfeld zu verfolgen. Die Positionen der Partikel werden gespeichert, um sie anschließend darzustellen. Die Parallelisierung dieser Visualisierungsmethode geschieht auf dieselbe Weise, wie es bei der Strömungsliniendarstellung der Fall ist.

Der wichtigste Unterschied ist jedoch die Darstellung. Hierfür wird die in Kapitel 7.1.3.6 vorgestellte `Particle` Klasse erweitert: Sie speichert nun zusätzlich zu jeder Position des einzelnen Partikels noch dessen gewählte Erscheinungsform, die gewählte Darstellungsgröße, Farbe und Textur. Durch die Kapselung der Darstellungsart in diese Klasse, kann theoretisch jeder dargestellte Partikel anders aussehen.

#### 7.1.4.1 Adaptive Streaklines

Adaptive Streaklines sind eine Erweiterung der Streaklines Tracing Methode. Durch sie ist es möglich, die Visualisierung der Partikel auf bestimmte Bereiche des Strömungsfeldes einzuschränken. Dies sind Bereiche, in denen das Strömungsfeld Turbulenzen (*Vorticity*) oder eine bestimmte Strömungsgeschwindigkeit (*Velocity*) aufweist. Diese Tracingmethode ist, bedingt durch den verwendeten Algorithmus, nur zusammen mit den Emissionsmethoden „Line“ und „Regular Field“ sinnvoll anwendbar.

Vorticity-Visualisierung

Das Auffinden von Turbulenzen im Strömungsfeld geschieht durch Berechnung und Vergleich der Partikelbahnen. Grundsätzlich werden zwei unterschiedliche Merkmale der Partikelbahnen überprüft. Das erste Merkmal ist die zeitliche Bahnveränderung des aktuellen Partikels. Um diese zu erfahren, wird die aktuelle Partikelposition mit dessen Position in den beiden vorangegangenen Zeitschritten verglichen. Das zweite zu überprüfende Merkmal ist die räumliche Bahnveränderung. Hier werden die Abweichungen zu den Flugbahnen benachbarter Partikel untersucht. Hierbei gilt für Emissionsart „Line“ eine $N2$-Nachbarschaft (zwei existierende Nachbarpartikel, die das Ergebnis beeinflussen) und für die Emissionsart „Regular Field“ eine $N4$-Nachbarschaft (vier existierende Nachbarpartikel, die das Ergebnis beeinflussen). Diese Nachbarschaftsverhältnisse sind in den Abbildungen 7.28 und 7.30 nochmals verdeutlicht. Sogenannte Randpartikel, also solche, die nicht die zur Berechnung notwenigen Nachbarn besitzen, werden grundsätzlich nur entsprechend der vorhandenen Nachbarn bzw. bei deren Nicht-Vorhandensein einzig bzgl. ihrer Zeitachse überprüft.

In Abbildung 7.28 wird veranschaulicht, wie die Verknüpfung zwischen den Partikeln einer adaptiven Streakline, emittiert aus einem Line-Injektor, aussieht. Bei der Initialisierung eines jeden Partikels bekommt der vorangegangene Partikel einen Pointer auf

den aktuellen Partikel und umgekehrt. Damit wird gewährleistet, dass jeder Partikel seine Nachbarn und deren Positionen kennt.

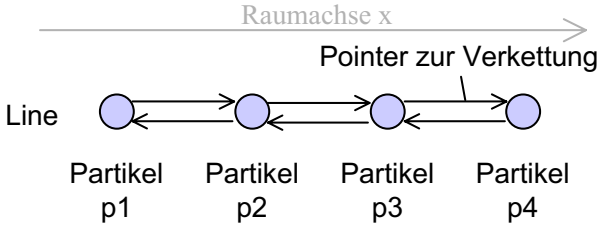



***Abbildung 7.28:*** *Initialisierung einer aus einem Line-Injektor emittierten adaptiven Streakline*

Das Ermitteln von Turbulenzen in einem mit dieser Emissionsmethode erzeugten Partikelfeld wird im Folgenden näher erläutert und in Abbildung 7.29 anhand von zwei möglichen Situationen veranschaulicht. Situation 1 zeigt den Vorgang der Kalkulation im Fall einer gleichmäßigen Strömung, während in Situation 2 eine Unregelmäßigkeit in der Strömung zu erkennen ist.

Ein mit einem Line-Injektor erzeugtes Partikelfeld besitzt eine Raumachse $x$ und eine Zeitachse $t$. Als Berechnungsgrundlage der zeitlichen Bahnveränderung werden die drei letzten Positionen des betroffenen Partikels verwendet. Zuerst werden die zwischen diesen Punkten liegenden Vektoren errechnet. Danach ist es möglich, einen Winkel zwischen diesen Vektoren zu ermitteln, dessen Größe die Flugbahnveränderung des Partikels beschreibt.

Da die Partikel ihre zurückgelegten Pfade selbständig speichern, ist es nicht notwendig, bei der Initialisierung besondere Vorbereitungen für diese Berechnung zu treffen. Um die räumliche Bahnveränderung zu ermitteln, werden zwei Vektoren auf der Raumachse ermittelt. Der erste Vektor liegt zwischen der gegenwärtigen Position von Partikel $p$ und der Position des vorangegangenen Partikels $p-1$. Der zweite Vektor wird beschrieben durch die aktuelle Position von Partikel $p$ und die Position des nachfolgenden Partikels $p+1$. Dieser zweite Vektor ist etwas aufwendiger in seiner Bestimmung, da zu diesem Zeitpunkt die Berechnung der neuen Position des Partikels $p+1$ noch nicht stattgefunden hat (siehe Abbildung 7.29, Schritt 1). Daher wird die Position mit einem Euler-Integrator approximiert. Die so erhaltene Position wird ausschließlich für diese eine Berechnung verwendet. Sollte nun einer der resultierenden Winkel einen bestimmten über die Benutzeroberfläche definierbaren Schwellenwert überschreiten, so wird der Partikel entsprechend der Größe des Winkels eingefärbt. Liegt der Winkel unterhalb des Schwellenwertes, so wird der Partikel nicht gezeichnet.

Die Vorgehensweise bei einer adaptiven Streakline und der gewählten Emissionsmethode Regular Field ist grundsätzlich sehr ähnlich, weicht jedoch dahingehend ab, dass jeder Partikel, der zusätzlichen Raumachse $y$ entsprechend, zwei weitere zu berücksichtigende Nachbarn besitzt. Der eine liegt auf der Raumachse $y$ über dem aktuellen Partikel und der andere darunter. Abbildung 7.30 veranschaulicht die Verknüpfung der

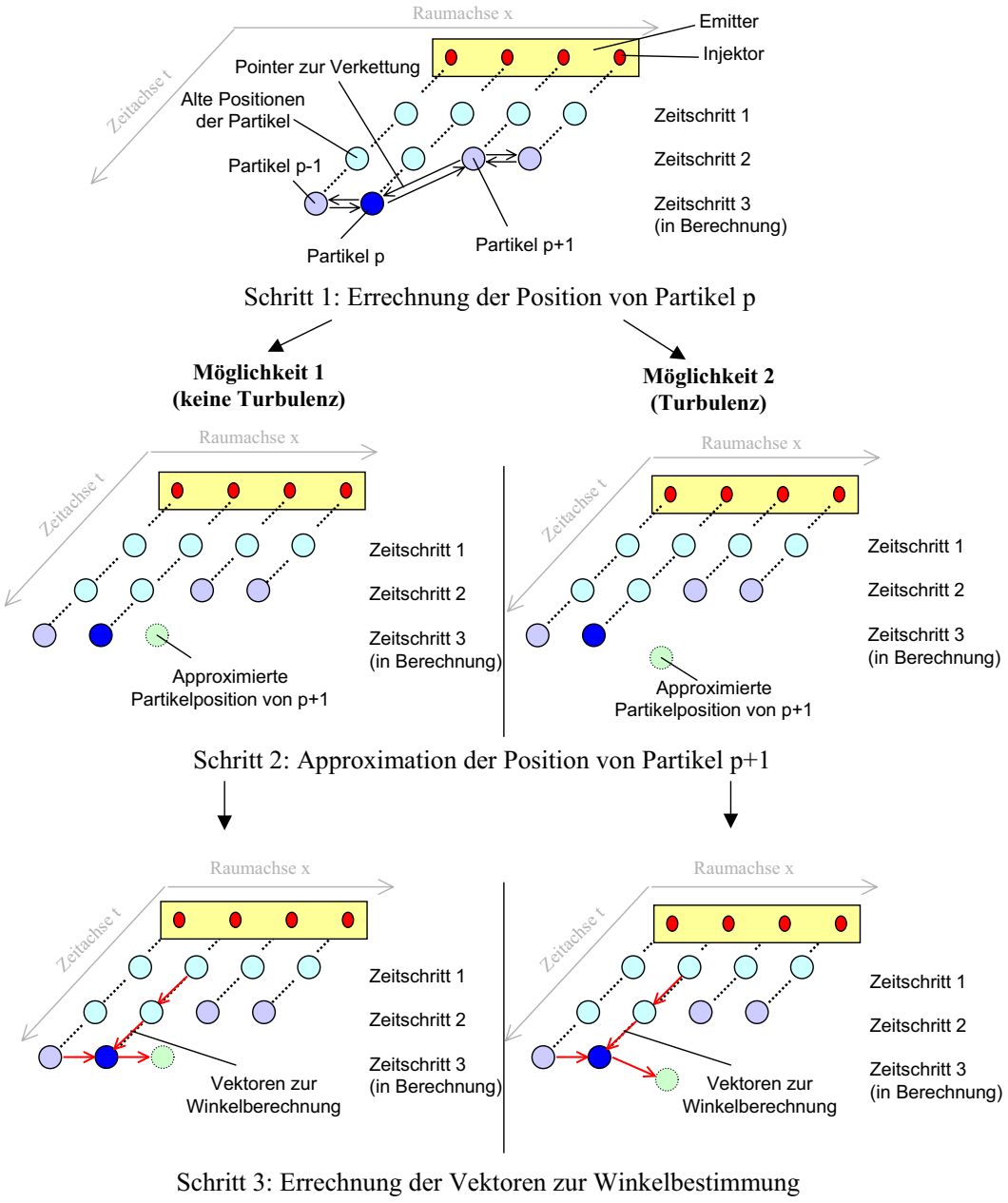



***Abbildung 7.29:*** *Berechnung einer aus einem Line-Injektor emittierten adaptiven Streakline*

Partikel in der Initialisierungsphase anhand eines Partikelfeldes mit einer vorgegebenden Höhe und Breite von jeweils vier Partikeln.

Durch die zusätzlichen Nachbarn ist es notwendig, eine zweite räumliche Winkelberechnung durchzuführen. Grundlage dieser Berechnung sind die beiden Vektoren zwischen der aktuellen Position von Partikel $p$ und der Position des darüber- bzw. darunterliegenden Partikels (in diesem Fall Partikel $p-4$ und $p+4$ - entsprechend der Breite des Partikelfeldes). Die Berechnung des Vektors zu Partikel $p+4$ ist wiederum nur möglich durch eine Approximation seiner Position nach dem gleichen Schema wie die

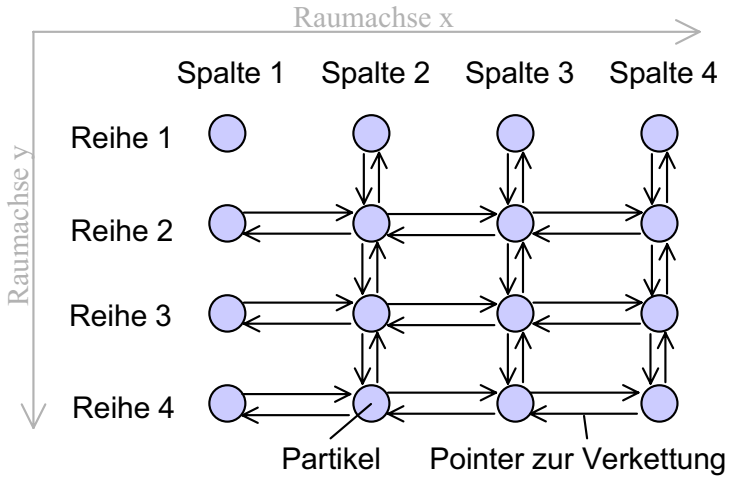



***Abbildung 7.30:*** *Initialisierung einer aus einem Field-Injektor emittierten adaptiven Streakline*

von Partikel $p+1$. Abbildung 7.31 verdeutlicht den Vorgang der Berechnung wiederum schrittweise anhand von zwei möglichen Entwicklungen.

Velocity-Visualisierung

Um Bereiche mit einer bestimmten Strömungsgeschwindigkeit zu visualisieren, wird nach jeder Aktualisierung der aktiven Partikel eine maximale Geschwindigkeit ermittelt, die der Geschwindigkeit des schnellsten Partikels entspricht. Der Anwender kann nun über die Benutzeroberfläche eine Geschwindigkeit bestimmen, die als Schwellenwert dient für die Entscheidung, ob ein Partikel gezeichnet werden soll. Dieser Schwellenwert wird in Prozent von der maximalen Geschwindigkeit ausgedrückt. Wird demnach als Schwellenwert zum Beispiel 100 Prozent gewählt, so wird kaum ein Partikel visualisiert, da die wenigsten Partikel den Maximalwert erreichen. Um diese Visualisierungsart zu ermöglichen, besitzt jeder Partikel eine Geschwindigkeit, die bei jeder Positionsaktualisierung neu errechnet wird.

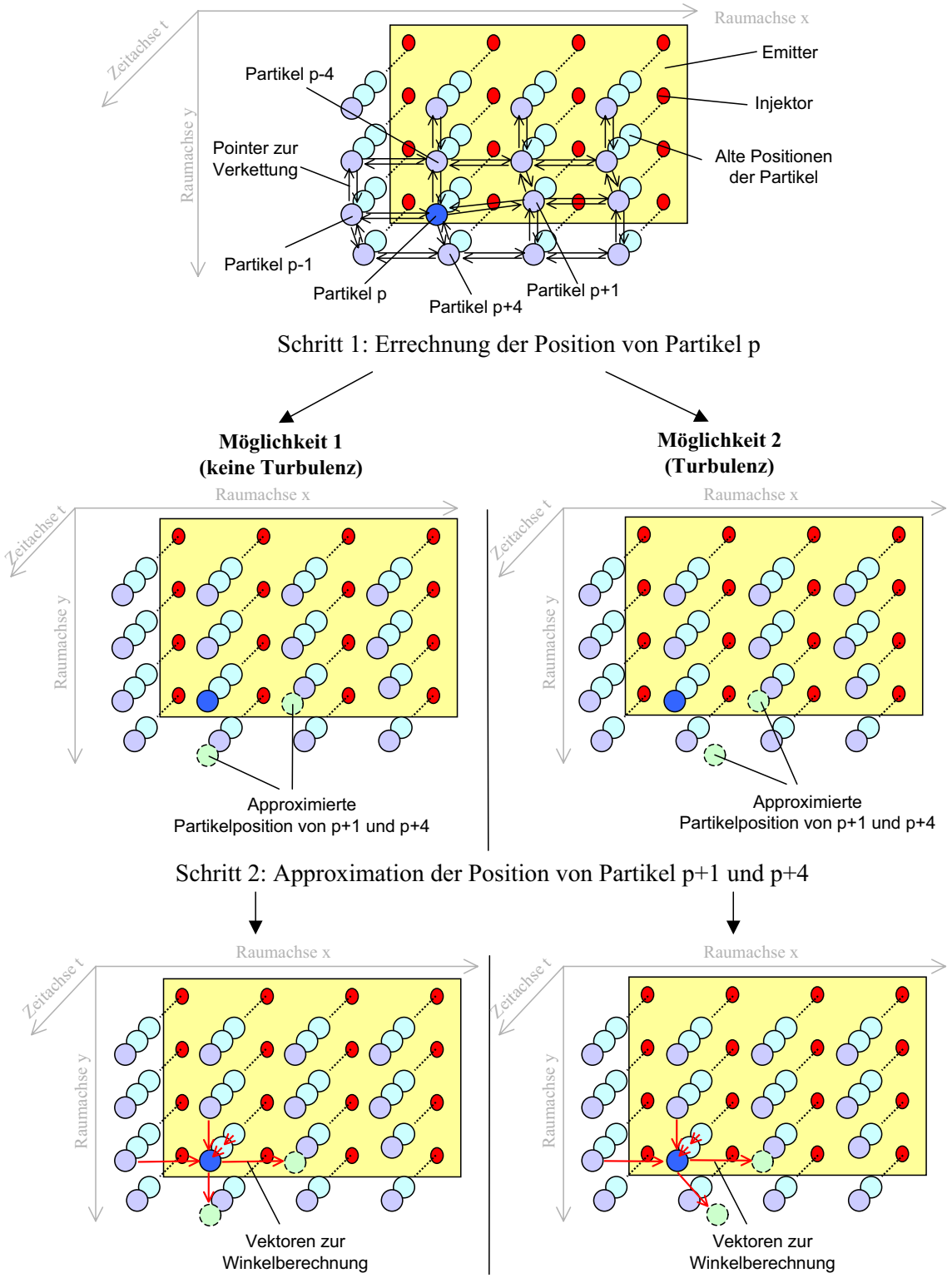



***Abbildung 7.31:*** *Berechnung einer aus einem Field-Injektor emittierten adaptiven Streakline*

#### 7.1.4.2 Darstellungs-Factory

Die Implementierung der unterschiedlichen Darstellungen für die Partikel geschieht mit Hilfe des Factory Konzepts. Abgeleitet von einer generischen Mutterklasse müssen sich einzelne Partikeldarstellungen bei dieser Factory registieren. Anschließend können die Partikel dann bei dieser Factory die zur Verfügung stehenden Darstellungsarten erfragen und verwenden.

Die Klasse `ParticleVisMod` dient als virtuelle Superklasse für die Klassen, die das Partikelrendering übernehmen. Jede Klasse, die von `ParticleVisMod` erbt, ist gezwungen, eine „draw()“-Funktion zu reimplementieren, in der die eigentliche graphische Darstellung stattfindet.

Die zugehörige Factory wird in die Klasse `ParticleEngine` integriert. Im vorliegenden Fall erhalten alle Partikel bei ihrer Erzeugung einen Zeiger auf ein Objekt der Klasse `ParticleVisMod`. Zum Aktivierungszeitpunkt wird diesem Zeiger dann die Instanz einer Subklasse von `ParticleVisMod` zugeordnet. Somit kennt der Partikel zwar die ihm zugewiesene Subklasse von `ParticleVisMod` nicht, kann aber bei einem Aufruf der „draw()“-Funktion die gewünschte Rendermethode verwenden.

Diese Strukturierung gewährleistet die komfortable Erweiterbarkeit der Darstellungsmöglichkeiten. Um eine neue Partikeldarstellungsart zu integrieren, reicht es aus, eine weitere Subklasse von `ParticleVisMod` zu erstellen und die entsprechende „draw()“-Funktion zu definieren. Veränderungen des übrigen Systems sind nicht notwendig.

Zur Zeit sind folgende Darstellungsmöglichkeiten für Partikel implementiert:

- Pixel
- Kugeln
- Würfel
- Streamlets
- Billboards

#### 7.1.4.3 Panel

Das zur Visualisierungsmethode passende Panel wird in Abbildung 7.32 gezeigt. Aufgebaut wie schon das Panel für Strömungslinien, bildet es eine Vielzahl von Einstellparametern ab, von denen hier nur die wichtigsten neu hinzugekommenen im Vergleich zum Panel der Strömungslinienvisualisierung (Kapitel 7.1.3.7) erläutert werden.

**PartcleVisModules**
: In diesem Bereich stellt der Benutzer ein, welche Erscheinungsform die Partikel in der Szene haben sollen. Alle ab diesem Zeitpunkt generierten Partikel erhalten die entsprechende Darstellung.

**Blending Mode & Farbbalken**
: Hier kann eingestellt werden, auf welche Art und Weise Transparenzwerte der Partikel mit der Umgebungsfarbe gemischt werden sollen. Außerdem kann die Grundfarbe der Partikel eingestellt werden.

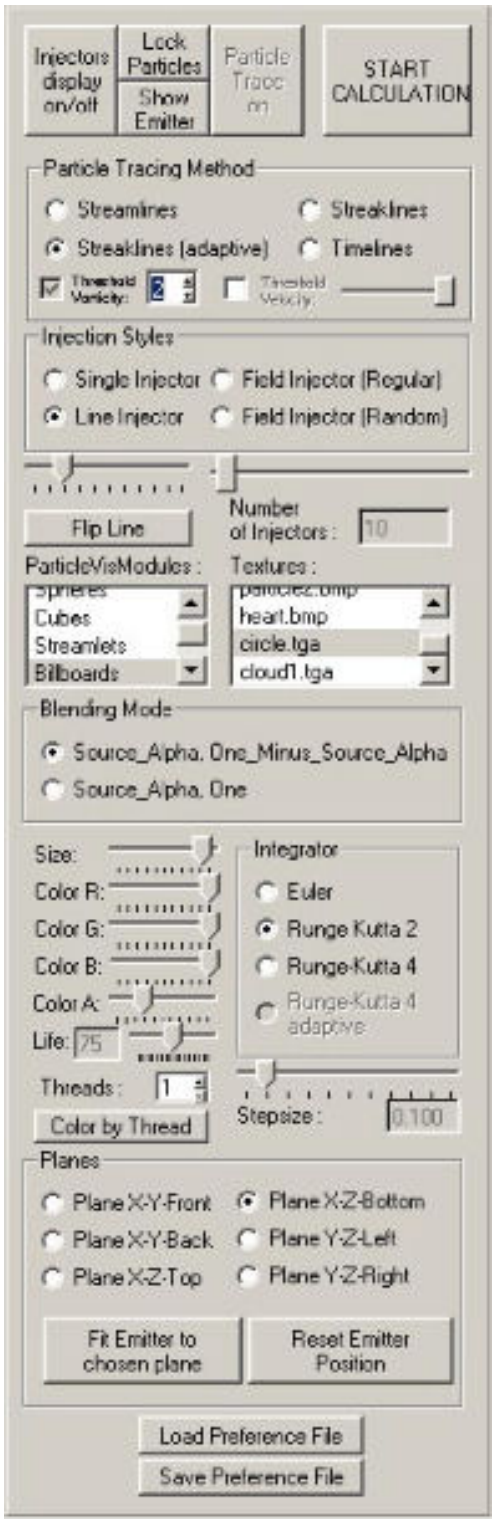



***Abbildung 7.32:*** *Panel zur Partikel Visualiesierungsmethode.*

#### 7.1.4.4 Klassenstruktur

Wie bereits erwähnt, unterscheidet sich die Klassenstruktur für den Aufbau der Partikelvisualisierungsmethode nicht allzu sehr von der der Strömungslinienvisualisierung. Der erkennbare Unterschied ist der zusätzliche Einbau der Darstellungsfactory und der zugehörigen Vererbungen der Klasse `ParticleVisMod` 7.33.

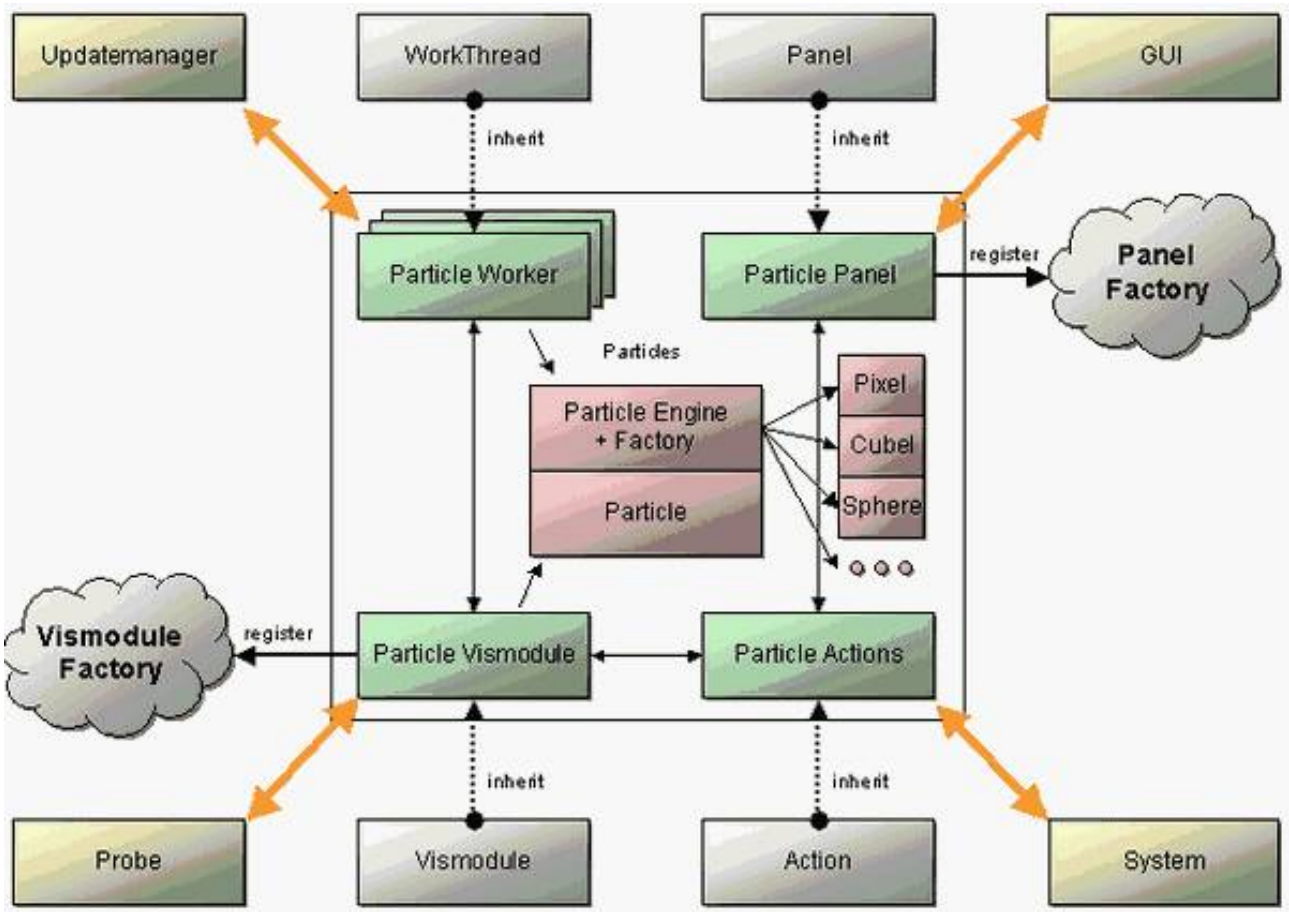



***Abbildung 7.33:*** *Die Klassen, die für die Implementierung des Partikel Verfahrens verwendet werden.*

#### 7.1.4.5 Partikelnebel

Partikelnebel ist ein Effekt, der durch Verwendung von Texturen, Farben und Blending und den adaptiven Streaklines, erzeugt werden kann. An folgendem Beispiel soll kurz veranschaulicht werden, wie dies geschieht.

Zu Anfang wird der Emitter so platziert, dass er eine Fläche der Bounding Box des Strömungsfeldes ausfüllt (siehe Abbildung 7.34). Dann wird ein Line Injektor gewählt und so angepaßt, dass die Partikel auf einer Position initialisiert werden, die eine möglichst gute Visualisierung des Strömungsfeldes gestattet. Die Strömung in diesem Beispiel bewegt sich von der hinteren oberen Kante zur vorderen unteren Kante.

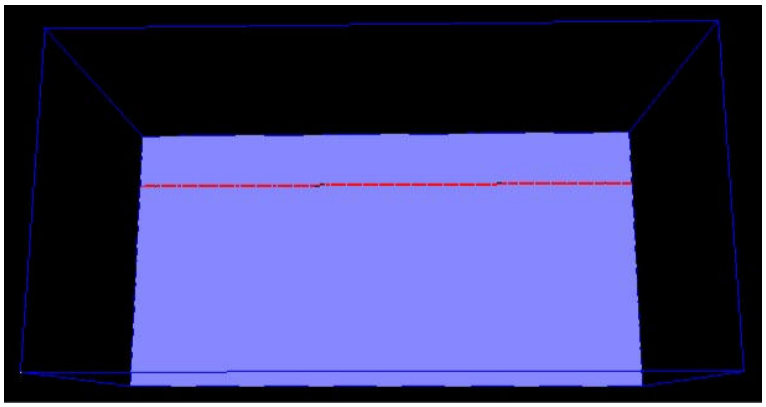


***Abbildung 7.34:*** *Entstehung von Partikelnebel - Schritt 1*

Nachdem der Emitter platziert ist, kann seine Darstellung aus der Szene entfernt werden. In Abbildung 7.35 ist nun zu sehen, wie durch die Verwendung von Streaklines eine Art Partikelteppich erzeugt wird, der die Strömung veranschaulicht. Als Partikeldarstellung werden hier zunächst Streamlets verwendet, um die Arbeitsweise von adaptiven Streaklines zu verdeutlichen. Die grüne Farbe der Partikel ist willkürlich gewählt.

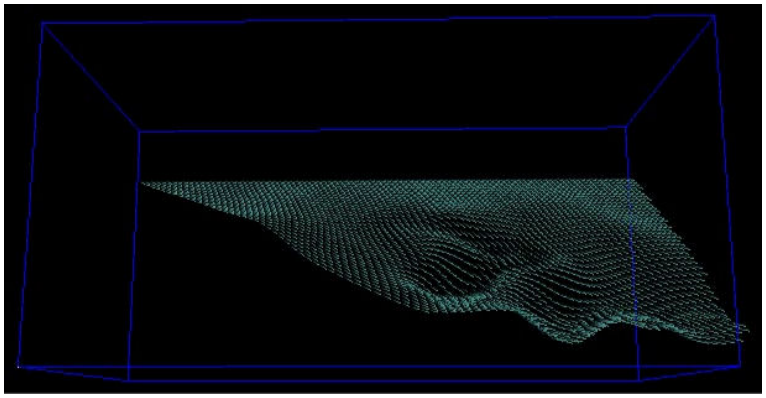


***Abbildung 7.35:*** *Entstehung von Partikelnebel - Schritt 2*

Im nächsten Schritt wird nun das Particle Tracing von Streaklines auf Adaptive Streaklines (Vorticity) umgestellt. In Abbildung 7.36 ist zu erkennen, wie an verschiedenen Stellen im Partikelteppich Löcher entstehen. Dort sind die Turbulenzen zu gering, um den Schwellenwert zu erreichen, der die Visualisierung der Partikel aktiviert.

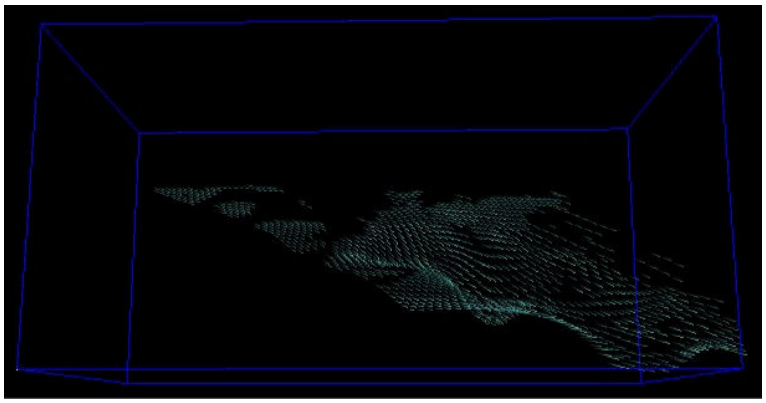


***Abbildung 7.36:*** *Entstehung von Partikelnebel - Schritt 3*

Verwendet man Billboards anstelle von Steamlets, so entsteht die Darstellung, die in Abbildung 7.37 zu sehen ist.

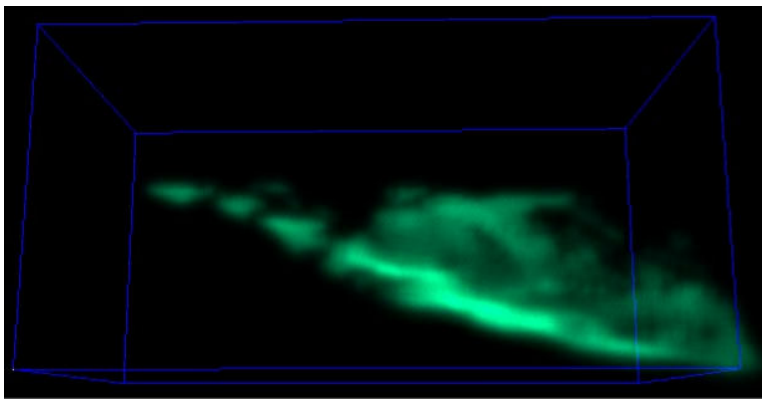
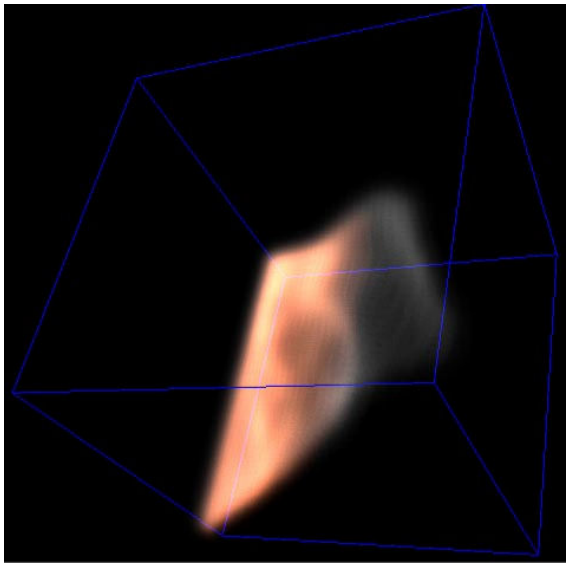


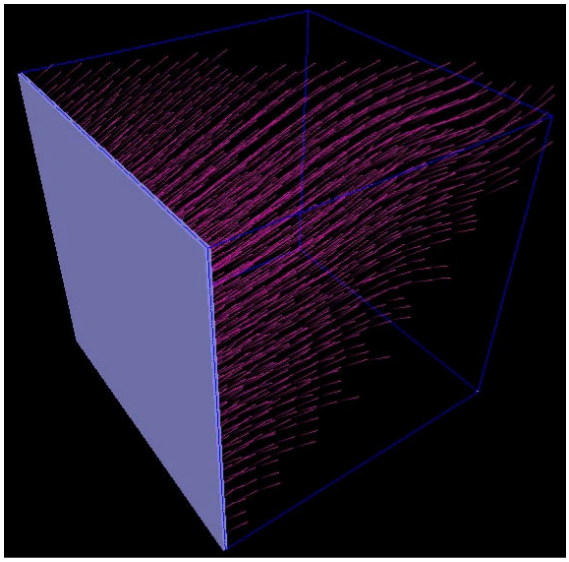
***Abbildung 7.37:*** *Entstehung von Partikelnebel - Schritt 4*

#### 7.1.4.6 Ergebnisbilder

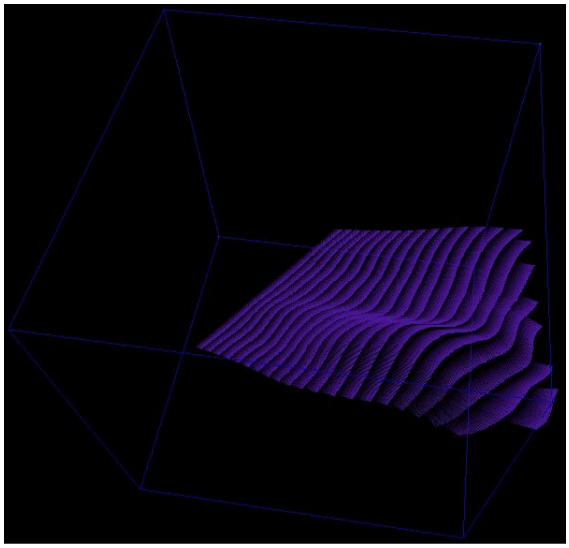
In Abbildung 7.38 werden einige weitere Ergebnisbilder der Partikel Visualierungsmethode präsentiert.



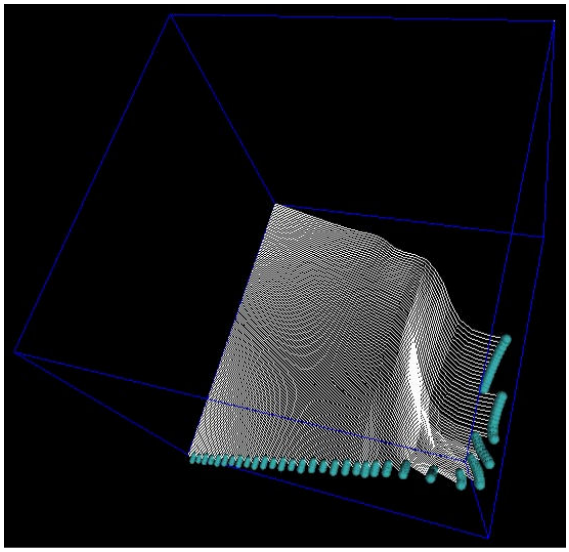
Visualisierung eines Feuereffektes mit Rauch. Verwendet wurden zwei Emitter mit unterschiedlichen Billboards.



Visualisierung eines Strömungsfeldes durch ein Partikelfeld.



Visualisierung eines Partikelteppichs durch Streaklines und Streamlets.



Visualisierung eines Partikelteppichs durch Timelines und Kugeln.

***Abbildung 7.38:*** *Verschiedene Parametereinstellungen und deren Resultate im Strömungslinien Verfahren.*

### 7.1.5 Parallel Shader Volume Rendering / Iso-Surfaces

Volumen-Rendering ist die Visualisierung von Daten, welche im dreidimensionalen Raum abgebildet werden. Hier werden ausschliesslich Skalarfelder betrachtet, welche als diskrete Daten vorliegen, und nicht z.B. als mathematische Funktion. Die Durchführung der Visualisierung ist i. A. sehr aufwendig, da jeder Punkt im Volumen zum endgültigen Bild beiträgt. Dies führte in der Vergangenheit zur Entwicklung vieler Rendering-Methoden, die unterschiedliche Kompromisse zwischen Rechenzeit und Bildqualität machen. Man unterteilt die möglichen Algorithmen hierfür in vier Gruppen (in Anlehnung an Van Gelder und Kim [VGK96]):

**Gruppe der image-order Algorithmen**
: Lösungen dieser Art berechnen das gesuchte Bild durch Verfolgung von Strahlen, deren Ausgangspunkt das Pixel auf dem Bildschirm ist. Dadurch, dass der Strahl mindestens solange verfolgt wird, bis sich der erreichte Farbwert nicht mehr weiter ändert, ist diese Methode sehr rechenintensiv und langwierig. Hierdurch wird allerdings die beste Bildqualität erreicht. Je nach Detailierungsgrad können hierbei auch physikalische Eigenschaften wie z.B. Refraktion und Spiegelung berücksichtigt werden. Zu dieser Gruppe gehört unter anderem das Ray-Casting oder Ray-Tracing Verfahren.

**Gruppe der object-order Algorithmen**
: Diese Methoden suchen eine polygonale Umsetzung des Skalarfeldes, um eine Flächenstruktur zu erhalten. Die durch Abtastung des Volumens entstandenden Polygon-Netze sind beliebig feine Approximationen des Originals, welche dann anschliessend gefüllt werden. Dadurch ist es möglich, die Rechenzeit auf Kosten der Bildqualität zu verkürzen. Methoden dieser Gruppe werden auch als *indirektes Volumen-Rendering* bezeichnet, da das Rendering selbst nicht mehr mit den Originaldaten stattfindet. Alle anderen Ansätze bezeichnet man als *direktes Volumen-Rendering*. Zu dieser Gruppe gehören z.B. das Marching Cubes oder Marching Tetraheder Verfahren.

**Shear-Warp Factorization**
: Durch Faktorisierung der Blick-Transformation in zwei Einzeltransformationen (eine Scherung und eine Verzerrung) kann das Bild mit Hilfe von Volumen-Scheiben (*slices*) berechnet werden. Shear-Warp kombiniert image-order und object-order Ansätze und bietet die Möglichkeiten der Parallelisierung und Scanline-Kompression. Siehe dazu [LL94] und [Lac96].

**Rendering mit Texturen**
: Hierbei wird die Texturfähigkeit der Graphikhardware (GPU) verwendet, um die Volumendaten selbst, oder vorberechnete Farbwerte zu speichern. Das Rendering geschieht i. A. durch Überlagerung von Flächen, welche durch die Texturierung Scheiben des Volumens repräsentieren (*slicing*).

Peter L. Williams und Nelson Max stellen in ihrer Arbeit von 1992 ([WM92]) ein optisches Modell zur Berechnung von Lichtausbreitungen in volumetrischen Dichteverteilungen vor. Darin werden das *Particle Model* und das *Continuous Model* als zwei Basismodelle definiert,

die das gleiche physikalische Phänomen auf unterschiedliche Weise interpretieren. Alle im Rahmen dieser Visualisierungsmethoden betrachteten Volumen-Rendering Methoden bilden den Effekt der Lichtausbreitung in Volumen mit Hilfe eines dieser Modelle nach.

Die Idee der Verwendung von 3D Texturen für das Volumen-Rendering wurde in der Literatur erstmals 1993 von Kurt Akeley erwähnt ([Ake93]). Er erkannte die Möglichkeit, die 3D Texturfähigkeit des RealityEngine Graphics Systems von SGI ([sgia]) einzusetzen. Die RealityEngine Architektur unterstützt hierzu non-mipmapped 3D Texturen mit bis zu 256x256x64 RGBA Texturpixel-Auflösung. Durch das Verwenden von 3D Texturen ist es möglich, die Texturpixel oder *Texels* (texel = texture-element) hardwarebeschleunigt tri-linear zu interpolieren. Timothy Cullip und Ulrich Neumann stellten im gleichen Jahr eine Methode zur Rekonstruktion von CT-Scan-Daten für die RealityEngine vor, und setzten somit die Idee in die Tat um ([CN93]).

Die Darstellung von beleuchteten Volumen gilt als besondere Schwierigkeit. Da bis vor kurzem keine Möglichkeit bestand, weiteren Einfluss auf die hardwareinterpolierten Daten zu nehmen, waren nur softwaretechnische, oder hardwaretechnische On-Chip Lösungen verfügbar ([VSSB95]). Für die Beleuchtungsrechnung werden die Gradienten bzw. die Flächennormalen des Volumens benötigt, welche in der 3D Textur kodiert werden können. Allen Van Gelder und Kwansik Kim untersuchten in ihrer Arbeit von 1996 ([VGK96]) unterschiedliche Architekturen für Renderingmethoden mit Beleuchtung.

Volumen-Rendering entwickelte sich gerade in medizinischen Bereichen zur Rekonstruktion von CT-Scan-Daten zu einem unverzichtbaren Hilfsmittel. So wurden grosse Anstrengungen unternommen, für diesen Einsatzbereich Virtual-Reality Tools mit interaktiven Bildraten zu entwickeln. Rüdiger Westermann und Thomas Ertl beschleunigten 1998 ([WE98]) die bisherigen Methoden durch intensiven Einsatz von Fragment Operationen wie z.B. *Stencil Test* oder *Alpha Test*. Beleuchtung wurde durch Definition einer Farb-Matrix erreicht. Allerdings setzten die meisten dieser Ansätze Multi-Pass Rendering voraus, d.h. das grafische Ergebnis ist erst nach mehreren Durchläufen (mit jeweils unterschiedlichen Einstellungen) vollständig.

1998 realisierten Dachille et al. eine neue Rendering Architektur ([DKC+98]). Die Interpolationseigenschaft der Graphikhardware wurde hierbei aufgrund ihre Leistungsfähigkeit verwendet, das Ergebnis wurde dann aber wieder in den Hauptspeicher übertragen. Das endgültige Bild entstand schliesslich mithilfe der *Sweep-Planes* Methode, mit der jede Scanline aus einer texturierten Fläche berechnet wird, die senkrecht zur Bildebene steht.

Zur Darstellung von grossen Datensätzen, deren Ausmaße über die Texturgrösse hinausgehen, schlugen LaMar et al. 1999 eine Multiresolution-Technik vor, welche interaktive Darstellungsraten erreicht ([LHJ99]).

Basierend auf dem Projected Tetrahedra Algorithmus (PT) von Shirley und Tuchman ([Sab88]) entwickelten Röttger et al. eine Methode für Tetraheder Volumen Zellen unter Verwendung von vorintegrierten 2D oder 3D Texturen ([RKT00]).

Die Einschränkungen lagen i. A. darin, dass nach einer Flächentexturierung keine weitere Anwender Operation im Rendering Prozess durchgeführt werden kann. Dies änderte sich durch die Einführung programmierbarer Graphik-Pipeline-Stufen für Standard PCs (Shader,

siehe Kapitel 3.6.5). Dadurch wurde es möglich, mit mehreren Texturen gleichzeitig vollwertige mathematische Berechnungen während der Rasterisierung pro Pixel durchzuführen. Die Industrie entwickelte zu Beginn unterschiedliche Realisierungsvarianten für Shader. Die Anwendung dieser Shader für Volumen-Rendering wurde von Resk-Salama et al. in Anlehnung an die *NVidia Register-Combiner Technik* untersucht ([RSEB+00]).

Klaus Engel et al. kombinierten 2001 die Methoden von Van Gelder und Kim, Röttger et al. und Resk-Salama et al. zu einem hochqualitativen, hardware-beschleunigten Rendering Tool ([EKE01]). Die vollständige Beleuchtungsrechnung wird dabei in Echtzeit auf der Hardware durchgeführt. Diese Arbeit lieferte den Ausgangspunkt für das hier entwickelte Visualisierungsverfahren und wird im Anschluss genauer erläutert und erweitert. Es gehört zur Kategorie der Textur basierten Volumen Rendering Verfahren.

#### 7.1.5.1 Klassifikation, Transferfunktion und Isofläche

Bevor ein Volumendatensatz dargestellt werden kann, müssen die darin enthaltenen Werte ihrer Bedeutung entsprechend klassifizert werden. Das Skalarfeld ist hierbei eine Aufnahme von Werten aus einem spezifischen Merkmalsraum. So ist z.B. ein CT-Scan-Datensatz eines Körperteils eine Menge von Dichtemessungen, durch die sich die Materialien voneinander abgrenzen lassen. Diese Zuordnung ist zwar nicht eindeutig, aber so kann z.B. Knochenmaterial von Haut unterschieden werden, da unterschiedliche Materialdichten vorliegen. Ein weiteres Beispiel ist die Wärmeverteilung innerhalb eines Raums. Der Merkmalsraum wird hierbei durch Temperatur charakterisiert. Durch Definition der Klassifikationvorschrift hat der Anwender die Möglichkeit, gezielt bestimmte Regionen im vorhandenen Wertebereich zu untersuchen. Eine Transferfunktion ist eine solche Vorschrift bzw. Zuordnung (Beispiel siehe Abbildung 7.39a). Zur grafischen Darstellung muss für jeden möglichen Messwert eine Abbildung auf die Farben Rot, Grün und Blau existieren. Weiterhin wird noch eine Abbildung auf einen Abschwächungskoeffizient benötigt, mit dem festgelegt wird, inwiefern das Material lichtdurchlässig ist. Physikalisch entspricht dieser Koeffizient der optischen Dichte. Analytisch betrachtet sind diese Transferfunktionen nicht-lineare Abbildungen, welche jedoch später für die Anwendung auf Rechnerbasis zu diskreten Tabellen reduziert werden.

Wird nun ein bestimmter Skalarwert, der *Iso-Wert* oder auch *Iso-Value*, auf volle Opazität abgebildet und alle anderen als transparent definiert, erhält man eine Isofläche (*Iso-Surface*), siehe Abbildung 7.39b und c). Die Flächen geben sehr schnell Aufschluss über die räumliche Struktur gleicher Werte, wenn gleichzeitig eine entsprechende Beleuchtung simuliert wird (*Shaded Iso-Surfaces*). Weiterhin können auch beiden Seiten einer Fläche unterschiedliche Farben zugeordnet werden, wodurch Innen und Aussen unterschieden werden kann. Siehe hierzu Abbildung 7.39d.

#### 7.1.5.2 Architekturen für 3D Textur Volumen-Rendering

Die Architektur definiert die Reihenfolge der Anwendung von Klassifikation, Interpolation und Schattierung (Shading) im Renderingprozess. Eingabe dabei sind immer die Daten, Gradienten und eine Klassifikationvorschrift. Die Ausgabe wird in den Framebuffer

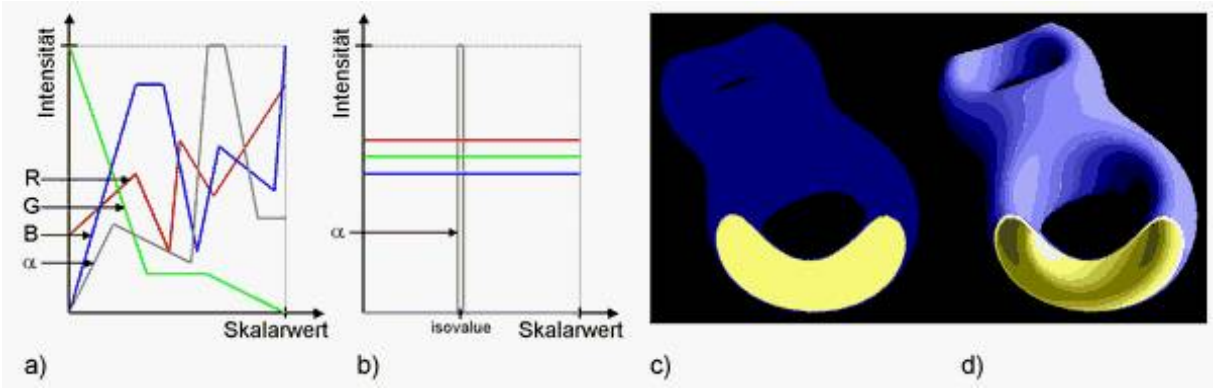



***Abbildung 7.39:*** *Ein Beispiel einer Transferfunktion (a) mit beliebigen Intensitätsverläufen für die Farbkanäle R,G,B und die Transparenz α. (b) zeigt die Transferfunktion einer Isofläche mit voller Opazität am Isowert. (c) und (d) zeigt eine gerenderte Isofläche nicht-schattiert und schattiert unter Verwendung verschiedener Farben für Innen und Außen.*

übertragen. Wird die Klassifikation vor der Interpolation durchgeführt, spricht man von *Pre-Classification*, d.h. die bereits klassifizierten Skalarwerte werden interpoliert. Den umgekehrten Fall bezeichnet man als *Post-Classification*.

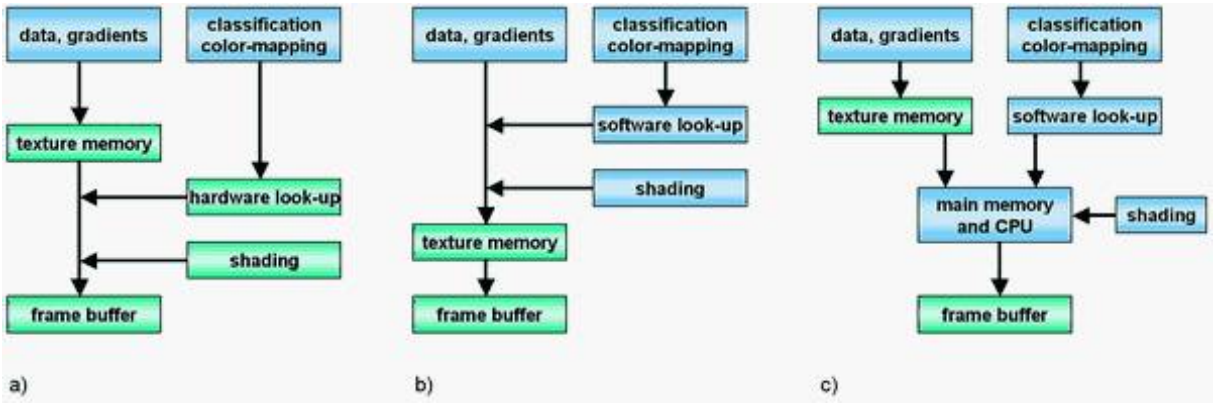



***Abbildung 7.40:*** *Architekturen für texturbasiertes Volumen-Rendering (grün: Hardware, blau: Software)*

Abbildung 7.40a ist mit Post-Classification die sog. „ideale Architektur“. In der ursprünglichen Architekur von Van Gelder und Kim ([VGK96], Abbildung 7.40b) werden die Daten vor der Texturgenerierung zugeordnet. Abbildung 7.40c schematisiert die Methode von Dachille et al. ([DKC+98]). Mit „Rückgriff“ auf Hauptspeicher und CPU wird hierdurch das gleiche Ergebnis, wie das der idealen Architektur erzielt.

Die Ausgaben des Texturspeichers sind tri-linear interpolierte Materialwerte, weswegen mit post-classification schärfere Abgrenzungen aufgelöst werden können. Pre-Classification resultiert dabei in unschärferen Bildern, da die bereits klassifizierten Materialien, bzw. die zugewiesenen Farben ineinander überblendet werden.

Der Zeitpunkt des Shadings definiert die Begriffe *pre-shading* und *post-shading*, wobei die

Beleuchtungsrechnung beim Letzteren nach der Klassifikation stattfindet. Pre-Shading hat den grossen Nachteil, das die Textur vollständig neu berechnet werden muss, wenn sich die Orientierung des Skalarfeldes bzgl. einer Lichtquelle oder des Betrachters ändert.

Durch die Eigenschaften der bisher auf dem Markt verfügbaren Hardware war die ideale Architektur nicht realisierbar. Dies kann jedoch durch die Entwicklung der Shadertechnologie nun erreicht werden.

#### 7.1.5.3 Slicing

Wird eine Geometrie zur grafischen Darstellung in einzelne Scheiben aufgeschnitten, spricht man vom Prinzip des Slicing. Durch die richtige Anordnung der Scheiben erhält man wieder die ursprüngliche Geometrie. Beim Volumen-Rendering nutzt man dieses Prinzip, um die Texturdaten einer 3D Textur sichtbar zu machen. Hierbei werden viele Flächen, bzw. Scheiben der Textur übereinander geschichtet (Analogie: Überlagerung von semi-transparenten Folien), wodurch ein räumliches Bild ensteht. Je nachdem ob 2D oder 3D Texturen verfügbar sind, werden hierfür unterschiedliche Techniken angewandt. Beim Volumen-Rendering mit Texturen ist der Abstand der slices bzw. deren Anzahl eine wichtige Kenngrösse.

Stehen nur 2D Texturen zur Vefügung, verwendet man *Object-Aligned Slices*, d.h. die Polygone (in diesem Fall nur Rechtecke) sind an dem Objekt ausgerichtet, welches das Skalarfeld beinhaltet. Das Volumen wird dazu in jeder Achsenrichtung ($X$, $Y$ und $Z$) in je einen Stapel mit 2D Texturdaten umgerechnet, wodurch man drei Flächenstapel erhält. Die Anordnung der texturierten Flächen und die Auswahl des Stapels ist abhängig von der Orientierung bzw. der Raumlage des Volumen-Rechtecks. Ab einem bestimmten Betrachtungswinkel wird der aktuelle Stapel und die zugehörigen Flächen durch neue ersetzt. Die einzelnen Rechtecke werden in back-to-front Reihenfolge gezeichnet, um ein entsprechendes Blending zu erreichen. Abbildung 7.41 zeigt die Anordnung aller drei Flächenstapel. Dadurch wird die eigentlich benötigte tri-lineare Interpolation durch bi-lineare substituiert.

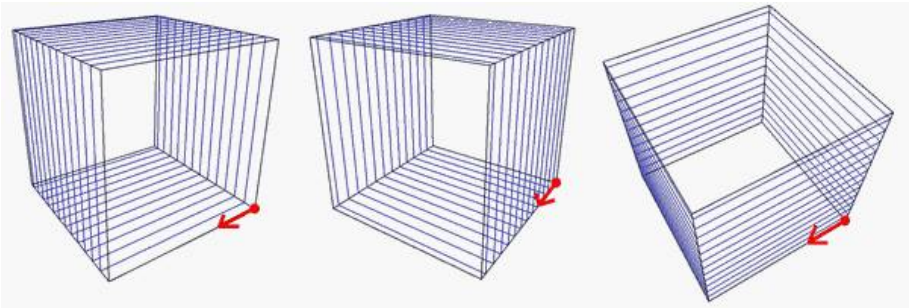


***Abbildung 7.41:*** *object-aligned slices bei 2D Texturen*

Durch Verwenden von 3D Texturen kann hardwaremäßige tri-lineare Interpolation auf Flächen parallel zur Bildebene ausgenutzt werden. Diese Flächen werden als *Viewport-Aligned Slices* bezeichnet (Abbildung 7.42). Das Clipping am Volumen-Rechteck liefert 3- bis 6-seitige Polygone, welche mit der 3D Textur korrekt gefüllt werden müssen. Dies geschieht durch die

Zuweisung von entsprechenden drei-dimensionalen Texturkoordinaten an jeden Knotenpunkt jedes Polygons. Die Koordinaten der Knotenpunkte werden hierzu in das Koordinatensystem der 3D Textur transformiert.

Eine schnelle Darstellung kann schon mit einfachen semi-transparenten Volumen gerendert werden (Abbildung 7.43). Wichtig ist, dass der Abstand zweier benachbarter Viewport-Aligned Slices bei perspektivischer Projektion konstant bleibt, währendessen er bei Object-Aligned Slices durch die Fluchtpunkt-Rechnung variiert. Dieser Abstand hat Einfluss auf die weiteren Berechnungen.

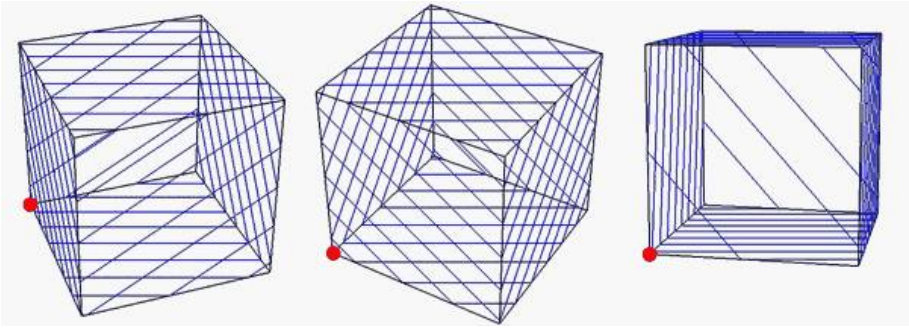


***Abbildung 7.42:*** *viewport-aligned slices bei 3D Texturen*

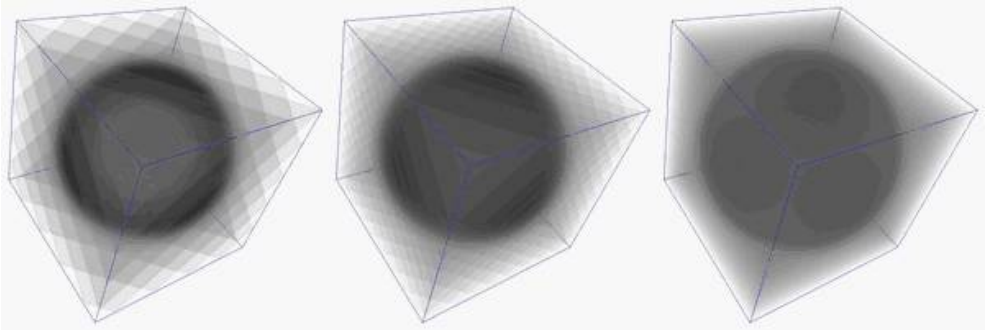


***Abbildung 7.43:*** *Einfache Volumen Darstellung mit 3D Texturen*

#### 7.1.5.4 Optisches Modell der Lichtausbreitung in Partikelfeldern

Die Aufgabe aller Volumen-Renderer ist es, die Integration aller Lichtmodulationen eines Sehstrahls innerhalb des Volumens für jeden Pixel zu berechnen. Dieses Integral wird als *Volume Rendering Integral* bezeichnet.

Eine exakte Simulation der Lichtausbreitung in Volumen ist sehr aufwendig und setzt die *Radiative Transport Theory* ([Kru90]) als Grundlage voraus. Für die hier entwickelte Visualisierung ist eine Approximation mit Hilfe eines einfacheren Modells völlig ausreichend. Aufgrund der Interpolationseigenschaft der 3D Texturhardware kann das *Continuous Model*, entwickelt von Williams und Max ([WM92]), näherungsweise realisiert werden.

Das Modell wurde aus dem *Particle Model* von Paolo Sabella ([Sab88]) entwickelt. Das Ergebnis seiner Arbeit ist das Volumen-Rendering Integral der Form:

$$I = c \int_{t_1}^{t_2} \rho(t) e^{-\tau \int_t^{t_2} \rho(u) \mathrm{d}u} \mathrm{d}t \tag{7.16}$$

mit den Bezeichnern:

| | |
|---|---|
| $I$ | totale Lichtintensität am Strahlende |
| $r$ | Radius der Partikel-Kugeln |
| $\kappa$ | Lichtintensität eines Partikels |
| $\tau$ | $= \frac{3}{4r}$ |
| $c$ | $= \kappa\tau$ |
| $t_i$ | Sehstrahlparameter |
| $\rho(t)$ | $= \rho(x(t), y(t), z(t))$, Dichtefunktion des Volumens |

Die Berechnung des Integrals ist sehr komplex, weswegen die Weiterentwicklung zum *Continuous Model* die Formel in je einen Teil für jede Farbkomponente unterteilt.

Das Volumen wird nun als leuchtende Gaswolke verstanden, in der jeder Punkt Licht aussendet und Licht absorbiert. Jeder Punkt hat daher zwei Eigenschaften: optische Dichte und Farbe. Durch die Wellenlängen-Abhängigkeit ist die optische Dichte durch $\rho(x, y, z, \lambda) \geq 0$, die emittierte Farbe durch $\kappa(x, y, z, \lambda)$ gegeben, mit $x, y, z$ als Koordinaten und $\lambda$ als Wellenlänge.

Wird das Skalarfeld durch $S(x, y, z)$ beschrieben, so lassen sich die beiden Eigenschaften durch die sechs Transferfunktionen $\rho_{red}$, $\rho_{green}$, $\rho_{blue}$ und $\kappa_{red}$, $\kappa_{green}$, $\kappa_{blue}$ formulieren. Dann ist z.B. $\rho(x, y, z, \lambda(red)) = \rho_{red}(S(x, y, z))$. Zur Modellentwicklung wird nun ein Sehstrahl $P(t)$ durch das Volumen angenommen, welcher über die Länge $t$ parameterisiert ist. Damit wird die endgültige Lichtintensität $I(t, \lambda)$, bzw. die endgültige Farbe, am Strahlende approximiert.

So erhält man

$$I(t_n, \lambda) = \kappa(\lambda) \underbrace{(1 - e^{-\rho(\lambda)(t_n - t_0)})}_{A} + I(t_0, \lambda) \underbrace{e^{-\rho(\lambda)(t_n - t_0)}}_{B} \tag{7.17}$$

Diese Gleichung beschreibt den *atop Operator* des Alpha-Blendings mit Blending-Parameter $\alpha = A$. In einem Abschnitt $[t_1, t_2]$ des Sehstrahls kann $\alpha = \rho(\lambda)(t_2 - t_1)$ als weitere Approximation angenommen werden.

Für das hier benötigte Verständnis der Anwendung der Transferfunktionen und der Bedeutung des Alpha-Blendings reicht dieser Einblick. Die vollständige Herleitung ist in [WM92] nachzulesen. Dieses Modell liefert die Grundlage für die Herleitung der hier vorgestellten Näherung.

#### 7.1.5.5 Pre-Integrated Volume Rendering

In vorangegangenen Abschnitt wurde ein optisches Modell zur Lichtausbreitung in volumetrischen Dichteverteilungen vorgestellt, das die resultierende Farbe eines parameterisierten Sehstrahls durch das Volumen approximativ bestimmt. Die Hauptidee der hier vorgestellten Visualisierungsmethode beruht nun darauf, diese Berechnung mit Hilfe von Viewport-Aligned Slices und 3D Texturen nachzubilden. Durch die Technik der Viewport-Slices wird der angenommene Sehstrahl in Strahlsegmente unterteilt, deren Farbe jeweils einen kleinen Beitrag zur gesamt-resultierenden Farbe des Sehstrahls leisten.

Um die Farben der Strahlsegmente nun durch die Dependent-Texture Fetch Technik hardwarebeschleunigt zu bestimmen, sind zunächst einige Näherungen des Volumen Integrals nötig. Schließlich wird der Framebuffer dazu verwendet, die endgültige Farbe eines Strahls zu bestimmen (Blending).

Zur schnellen numerischen Berechnung des Volumen-Rendering Integrals (Gleichung 7.16) müssen entsprechende Vereinfachungen herangezogen werden. Ziel ist es, die analytischen Integral-Ausdrücke durch Summen oder Produkte zu ersetzen, um die Berechnungen dann auf Rechner- und Hardwarebasis durchführen zu können. In Anlehnung an das vorgestellte optische Modell und den Arbeiten in [EKE01] kann das Integral, ausgedrückt mit Farbe (*color*) und optischer Dichte (*extinction*), in die Form

$$I = \int_0^D color(\mathbf{x}(t)) e^{-\int_0^t extinction(\mathbf{x}(t'))\mathrm{d}t'} \mathrm{d}t$$

gebracht werden. Dabei bezeichnen

- $I$ totale Lichtintensität am Strahlende
- $\mathbf{x}$ Sehstrahl
- $t$ Distanz zum Augpunkt
- $D$ maximale Distanz, d.h. $color(\mathbf{x}(n)), n > D = 0$

Durch die Beschreibung des Skalarfeldes mit $s(\mathbf{x})$, sowie durch die Spezifikation der Transferfunktionen für Farben mit $\vec{c}(s)$ und optischer Dichte mit $\tau(s)$ in Abhängigkeit des Skalarwertes $s$, erhält man

$$I = \int_0^D \vec{c}(s(\mathbf{x}(t))) e^{-\int_0^t \tau(s(\mathbf{x}(t')))\mathrm{d}t'} \mathrm{d}t.$$

Die Funktion $\vec{c}(s)$ liefert ein Vektor im RBG-Raum, entsprechend den drei Transferfunktionen $\kappa$ im vorangegangenen Abschnitt des Optischen Modells. Die dort eingeführten Bezeichnungen für die optische Dichte $\rho_{red}, \rho_{green}, \rho_{blue}$ werden nun durch die skalare Funktion $\tau(s)$ repräsentiert, wobei diese Funktion jetzt für alle drei Farbkanäle gilt.

Die viewport-aligned slices Rendering Methode liefert äquidistante Flächen im Skalarfeld, das in einer 3D Textur gespeichert ist. Die Unterteilung des Sehstrahls durch die Flächen schematisiert Abbildung 7.44.

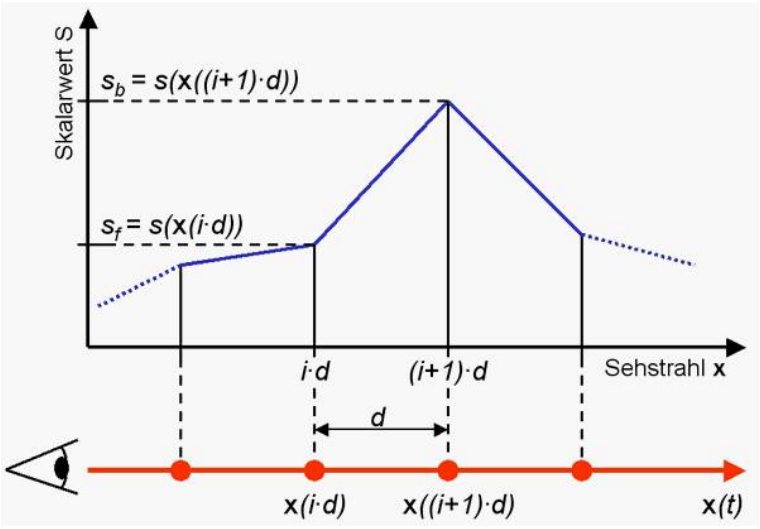



***Abbildung 7.44:*** *Strahlsegmente entlang eines Sehstrahls durch das Skalarfeld*

Der Sehstrahl durchläuft die viewport-aligned Slices im Abstand $d$, welcher unabhängig von der Orientierung des Volumens ist, und daher als konstant angenommen wird. Diese Eigenschaft gilt nicht bei der Methode für 2D Texturen oder der Projected Tetrahedra, da die Länge des Strahlsegmentes durch die perspektivische Betrachtung, bzw. durch die Geometrie eines Tetraeders nicht konstant bleibt. Das hier aufgestellte Konzept ist daher einfacher und platzsparender zu realisieren.

Das Integral der $e$-Funktion kann nun durch die Riemann Summe über $n$ gleiche Strahlsegmente der Länge $d = D/n$ ersetzt werden. Wegen besserer Lesbarkeit wird $e^{(\dots)}$ nun durch $exp(\dots)$ ausgedrückt. Siehe hierzu auch [EKE01].

$$exp\left(-\int_0^t \tau\left(s\left(\mathbf{x}\left(t'\right)\right)\right) \mathrm{d}t'\right) \approx exp\left(-\sum_{i=0}^{t/d} \tau\left(s\left(\mathbf{x}\left(id\right)\right)\right) d\right)$$

$$= \prod_{i=0}^{t/d} exp\left(-\tau\left(s\left(\mathbf{x}\left(id\right)\right)\right) d\right) = \prod_{i=0}^{t/d} \left(1-\alpha_i\right)$$

Hiermit ist die Opazität des $i$-ten Strahlsegments also durch

$$\alpha_i \approx 1 - exp\left(-\tau\left(s\left(\mathbf{x}\left(id\right)\right)\right) d\right)$$

approximiert, oder weiter genähert durch

$$\alpha_i \approx \tau\left(s\left(\mathbf{x}\left(id\right)\right)\right).$$

Die Transparenz ist damit $1 - \alpha_i$. Wird die emittierte Farbe $\vec{C}_i$ des $i$-ten Strahlsegments durch $\tau(s(\mathbf{x}(id))) \cdot \vec{c}(s(\mathbf{x}(id)))d$ beschrieben, erhält man das Volumen-Rendering Integral in der Form

$$I \approx \sum_{i=0}^{n} \vec{C}_i \prod_{j=0}^{i-1}(1 - \alpha_j)$$

Durch Anwendung des *atop*-Operators (siehe Gleichung 7.17), bzw. back-to-front Blending erhält man

$$\vec{C}'_i = \vec{C}_i + (1 - \alpha_i)\vec{C}'_{i+1} \tag{7.18}$$

mit $\vec{C}'_i$ als aufsummierte Farbe des $i$-ten Strahlsegments. Das back-to-front Blending kann hardwarebeschleunigt durch die Blending-Funktion `GL_SRC_ONE_MINUS_ALPHA` des Framebuffers umgesetzt werden.

#### 7.1.5.6 Pre-integrated Classification

Der letzte Ausdruck (Gleichung 7.18) ist auf Rechnerbasis realisierbar. Um den Vorgang zu beschleunigen, berechnet man die einzelnen Beiträge aus $\vec{C}_i$ und $\alpha_i$ der Strahlsegmente schon im Voraus. Dieses Verfahren wird als Pre-integrated Classification bezeichnet. Die einzigen Variablen bei der Berechnung des Alpha- und des Farbwerts sind $s_f$, $s_b$ und $d$, also die Skalarwerte zu Anfang und Ende des Segments sowie die Länge des Segments. Da mit konstanten Segmentlängen gerechnet werden kann, ergibt sich nur eine Abhängigkeit von $s_f$ und $s_b$ (siehe [EKE01]):

$$\begin{aligned}\alpha_i(s_f, s_b) &= 1 - exp\left(-\int_{id}^{(i+1)d} \tau(s(\mathbf{x}(t)))\mathrm{d}t\right) \\ &\approx 1 - exp\left(-\int_0^1 \tau((1-\omega)s_f + \omega s_b)d\mathrm{d}\omega\right),\end{aligned}$$

entsprechend für die Farben:

$$\begin{aligned}\vec{C}_i(s_f, s_b) \approx & \int_0^1 \tau\left((1-\omega)s_f + \omega s_b\right)\vec{c}\left((1-\omega)s_f + \omega s_b\right) \\ & \cdot exp\left(-\int_0^\omega \tau\left((1-\beta)s_f + \beta s_b\right)d\mathrm{d}\beta\right)d\mathrm{d}\omega.\end{aligned}$$

Hierbei wird angenommen, das es sich um ein stückweise lineares Skalarfeld handelt (Interpolation mit $\omega$ und $\beta$).

Die Parameter-Abhängigkeit von $s_f$ und $s_b$ ist dabei minimal. Das bedeutet aber gleichzeitig, dass jede Änderung der Transferfunktionen eine Neuberechnung der $\alpha$- und Farbwerte

nach sich zieht. Zur wissenschaftlichen Evaluierung eines unbekannten Skalarfeldes sollte das System jedoch bei Änderung der Transferfunktionen schnell reagieren. Diese Vor-Integration der Klassifikation kann durch vorberechnete Integraltabellen für $\tau(s)$ mit $T(s) = \int_0^s \tau(s)\mathrm{d}s$ und für $\tau(s)\vec{c}(s)$ mit $K(s) = \int_0^s \tau(s)\vec{c}(s)\mathrm{d}s$ beschleunigt werden. Man erhält die einfachen Ausdrücke

$$\alpha(s_f, s_b) \approx 1 - exp\left(-\frac{d}{s_f - s_b}(T(s_b) - T(s_f))\right) \qquad \text{und} \tag{7.19}$$

$$\vec{C}(s_f, s_b) \approx \frac{d}{s_b - s_f}(K(s_b) - K(s_f)). \tag{7.20}$$

#### 7.1.5.7 Slab-by-Slab Rendering

Im Gegensatz zu den früheren Methoden für Volumenvisualisierung mit 3D Texturen, welche die Fläche als solche mit den Texturdaten füllen und übereinander schichten (*Slice-By-Slice*), wird durch die Vorberechnung der Segmente nun ein *Slab-By-Slab* Rendering erzielt. Ein *Slab* bezeichnet den Teilbereich des Skalarfeldes, welcher von zwei Flächen (slices) eingeschlossen wird (siehe Abbildung 7.45). Für das Rendering eines Slabs müssen daher beide Flächen innerhalb der 3D Textur (Vorder- und Rückseite) und deren Abstand bekannt sein. Die Darstellung einer der beiden Seiten repräsentiert durch eine entsprechende Texturierung die im kompletten Slab angenommene Lichtenergie.

Die Dicke der Slabs, bzw. der Abstand der Slices, entspricht der verwendeten Sampling-Rate des Skalarfeldes in Richtung des Sehstrahls. Wie bereits erwähnt, wird das Skalarfeld zwischen den Flächen als linear angenommen (siehe auch Abbildung 7.46). Ebenso interpoliert die Texturhardware die in der 3D Textur befindlichen Skalarwerte. Bei einem Abstand, der grösser als die zugrundeliegende Pixelauflösung der 3D Textur ist, können Details verloren gehen (siehe Abbildung 7.46a). Um also alle gespeicherten Skalarwerte richtig zu erfassen, muss der Abstand kleiner oder gleich der Pixelauflösung der Textur gewählt werden (siehe Abbildung 7.46b).

Durch die Änderung der Flächendistanz ändert sich gleichzeitig die Gesamtzahl der Flächen, welche benötigt werden, um den darzustellenden Bereich vollständig abzudecken. Diese Anzahl ist proportional zur Zeit, die die Hardware für das Rendering benötigt, da jede Fläche von der GPU texturiert werden muss.

#### 7.1.5.8 Bricking

Wie in Kapitel 3.6.4 beschrieben, haben Texturen eine maximal mögliche Auflösung, die von der Graphikhardware unterstützt wird. Um nun auch Datenmengen mit 3D Texturen abbilden zu können, deren Auflösung die maximal hardwareseitig unterstützte überschreitet, wird die darzustellende Gesamtdatenmenge unterteilt. Dieses Verfahren wird als Bricking bezeichnet. Um Details des Skalarfeldes nicht durch Subsampling zu verlieren, wird für jeden

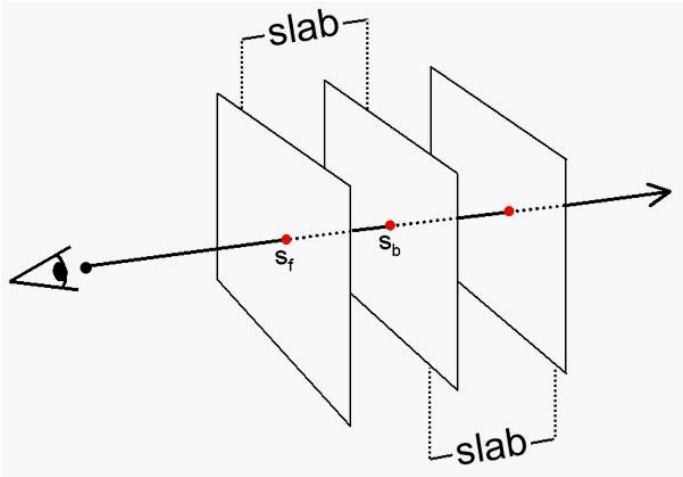



***Abbildung 7.45:*** *Slab-By-Slab Rendering mit Vorder- und Rückseite*

Frame der sondierte Teilbereich in kleine Quader (*Bricks*) gleicher Grösse zerlegt. Jeder Brick entspricht dabei einer 3D Textur.

Die Anzahl der Bricks, sowie die gewünschte Texturauflösung sind in allen drei Raumachsen durch *Brickcount* und *Brickresolution* frei definierbar, wobei die Texturauflösung aufgrund der Spezifikation der Graphikhardware einer Zweierpotenz $2^i$ entspechen muss. Durch Bricking kann eine Gesamtauflösung des Skalarfeldes erreicht werden, die grösser ist, als das hardwareseitig unterstützte Texturgrössenlimit. Durch die Möglichkeit, die urspüngliche Gitterauflösung des in der Datenquelle gespeicherten Skalarfeldes abzufragen, werden die Anzahl der Bricks und deren Texturauflösungen optimal initialisiert.

> *Beispiel*: Man erhält bei einem Bricking von 3x3x2 Bricks und 150 Zeitschritten insgesamt 2700 3D Texturen. Bei einer Texturauflösung von 32x32x32 und Platzbedarf von 4 Bytes pro Texel sind dies ca. 338 MB. Diese Zahl relativiert sich allerdings ein wenig, falls die Graphikhardware die Technik der Textur Kompression unterstützt.

Zur Beschreibung der Bricks wird $Bricksize \in \mathbf{R}^3$ und $Brick.index \in \mathbf{N}^3$ verwendet. Der Index bezeichnet die Position eines Bricks innerhalb eines sondierten Bereiches der Datenquelle. Für jeden Frame der Datenquelle bleibt die Brickunterteilung des durch die Sonde abgedeckten Datenbereiches identisch, da die Grösse aller Frames konstant ist, und sich die Lage und die Grösse der Sonde über die Frames ebenfalls nicht ändert. Die Position $\vec{P}$ eines Bricks mit Index $I$ im Raum wird dann durch

$$\vec{P} = \text{DataSource.start} + \text{Probe.position} + I * \text{Bricksize} \tag{7.21}$$

bestimmt.

Abbildung 7.47 zeigt ein mögliches Bricking innerhalb eines Teilbereichs des Volumens.

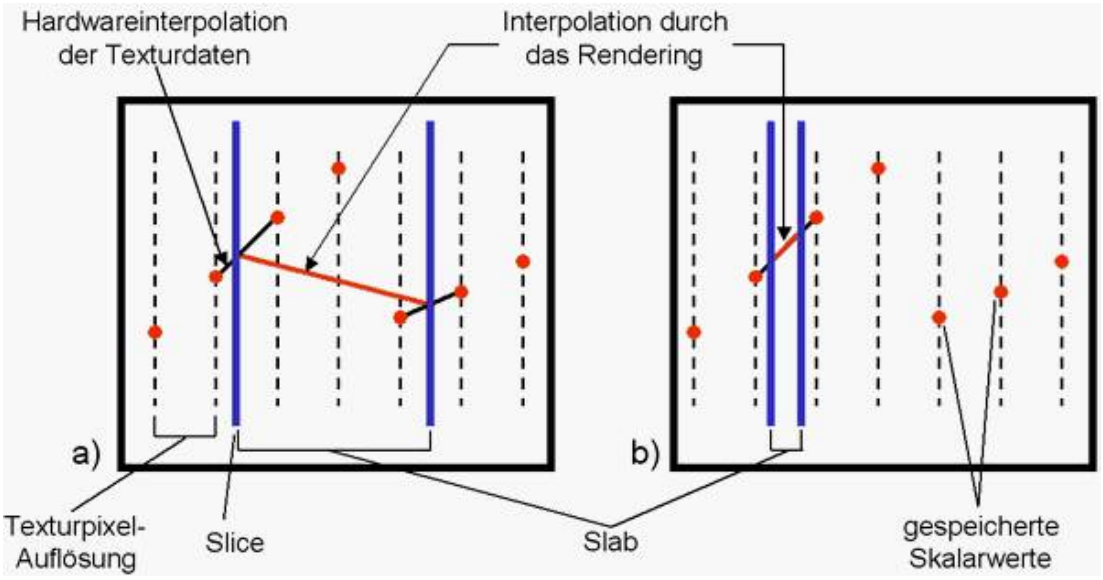



***Abbildung 7.46:*** *Das Sampling bei Slab-by-Slab Rendering. In Darstellung a) ist der Abstand der Slices zu gross, wodurch Details verloren gehen können. Darstellung b) zeigt das Sampling bei kleineren Abständen.*

Das Erstellen der 3D Texturen ist für beide Visualisierungsmethoden (Volumen Rendering und Iso-Flächen Darstellung) als Vorberechnungsschritt erforderlich. Der Inhalt der Texturen hängt dabei von der gewählten Visualisierungsmethode, bzw. von der Verfügbarkeit der `GL_ARB_fragment_program` Extension ab.

Wird die Brickunterteilung des sondierten Skalarfeldes der Datenquelle auch für die Einteilung der Texturdaten verwendet, führt dies zu Fehlern bei der Zusammensetzung der einzelnen Bricks zur vollständigen Sonde. Dies liegt daran, dass durch die Slice-Projektion Texturkoordinaten für die Rückseite des Slabs entstehen können, welche ausserhalb der definierten 3D Textur liegen. Je nach Paddingeinstellung der Textur-Hardware (vgl. [WJS99]) liefert das Sampling dann entweder eine Fortsetzung der Texels vom Rand der Textur (clamp), einen konstanten Wert (border-color) oder eine Wiederholung (repeat) der vollständigen Textur. Da die Hardware-Interpolation beim Auslesen von Texturdaten über deren Rand hinaus nicht auf die Daten der entsprechenden Nachbartextur zugreifen kann, muss dafür gesorgt werden, dass sich alle Texels am Rand der Textur mit den Nachbartexturen im entsprechenden Datenfenster überschneiden. Beim Zeichnen der Flächen, bzw. der Slabs müssen jedoch die Texturkoordinaten an diese Überschneidung angepasst werden. Die Anpassung erreicht man, indem man alle Texturkoordinaten auf einen etwas kleineren Quader abbildet. Die Korrektur der Texturkoordinaten ist abhängig von der Auflösung der Textur und wird durch die Skalierung $S_c$ und Translation $T_c$ durchgeführt:

$$S_c = 1 - \begin{bmatrix} \frac{2}{Brickresolution.x} \\ \frac{2}{Brickresolution.y} \\ \frac{2}{Brickresolution.z} \end{bmatrix} \tag{7.22}$$

$$T_c = \begin{bmatrix} \frac{1}{Brickresolution.x} \\ \frac{1}{Brickresolution.y} \\ \frac{1}{Brickresolution.z} \end{bmatrix} \tag{7.23}$$

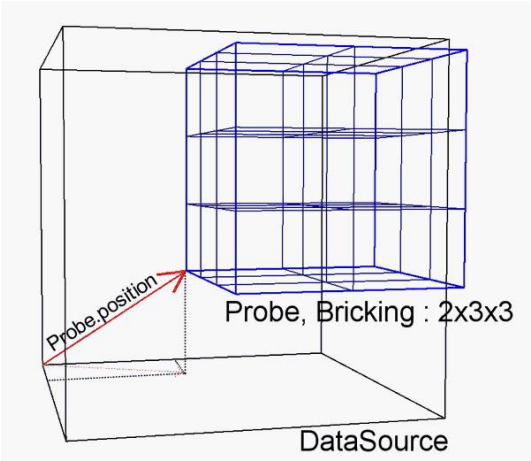



***Abbildung 7.47:*** *Beispiel einer Sonde mit eingezeichnetem Bricking*

Abbildung 7.48a zeigt das Interpolationsverhalten der Texturhardware (ein-dimensional). Der mögliche Fehler ergibt sich am Rand der Textur, da die korrekten Nachbartexels nicht bekannt sind. Die Überlappung der Randbereiche, um die richtige Interpolation der Texturdaten zu erhalten, zeigt Abbildung 7.48b (zwei-dimensional).

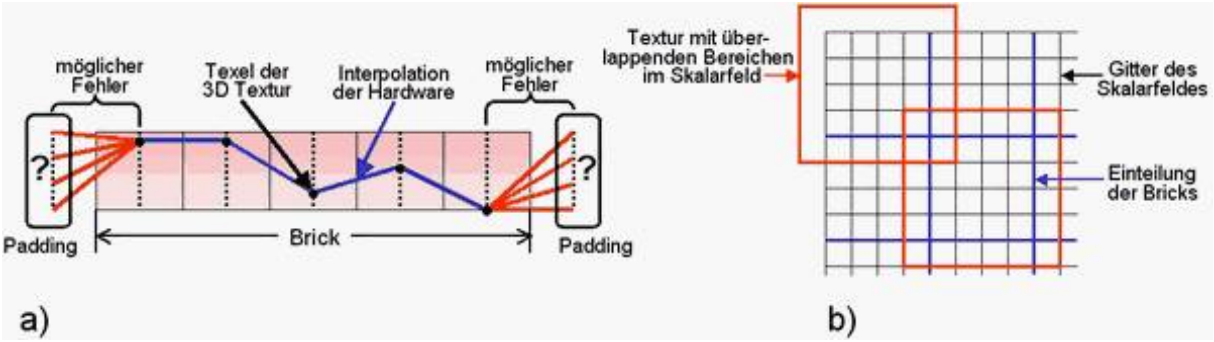



***Abbildung 7.48:*** *Der mögliche Fehler am Rand der Bricks entsteht durch die falsche Interpolation der Texturdaten (a). Abbildung (b) zeigt den Datenbereich der Texturen, der sich im Skalarfeld mit den Nachbartexturen überlappt.*

Die Verwendung von Texturen unter OpenGL wurde ausschliesslich für Farb-Texturen vorgesehen, d.h. alle Texturdaten werden auf den Zahlenbereich für Farben ($[0.0, \ldots, 1.0]$) begrenzt. Werden die normierten Gradienten, welche im Intervall $[-1.0, \ldots, 1.0]$ liegen in der Textur gespeichert, müssen sie vorher auf das Intervall $[0.0, \ldots, 1.0]$ normiert werden. Man bezeichnet diese Skalierung und Verschiebung als *Bias-Shift*. Vor dem Verwenden der Gradienten im Fragment Shader muss diese Normalisierung wieder rückgängig gemacht werden. Die gleiche Art der Normierung muss auch für die 2D IP-Textur (siehe Kapitel 7.1.5.12) berücksichtigt werden, da die Interpolationsgewichte ebenfalls im Intervall $[-1.0, \ldots, 1.0]$ liegen. Wird statt dessen ein Fragment Programm verwendet, können diese Gradienten automatisch on the Fly erstellt werden. Hier wird kein Bias-Shift mehr benötigt.

#### 7.1.5.9 Parallelisierung

Das Füllen der Texturspeicher lässt sich auf zwei Ebenen folgendermassen parallelisieren:

**Ebene 1: Frames**

Die einzelnen Frames des Datensatzes werden Frame-by-Frame sequentiell berechnet. Dies hat den Vorteil, das vollständig berechnete Frames bereits angezeigt werden können, während die übrigen Frames noch in Arbeit sind. Wird der Datensatz als Animation dargestellt, werden nur die anzeigebereiten Frames in der Animationssequenz verwendet. Abbildung 7.49 schematisiert den sequentiellen Zyklus der Berechnung bei gleichzeitiger Darstellung von bereits verfügbaren Frames. Die Parallelisierung durch Threads findet unabhängig von der Brickanzahl statt. Das bringt den Vorteil, dass, auch wenn die Anzahl der Threads grösser als die Anzahl der Bricks ist, die Laufzeit der Anwendung mit der Anzahl der verfügbaren parallel-arbeitenden CPUs besser skaliert.

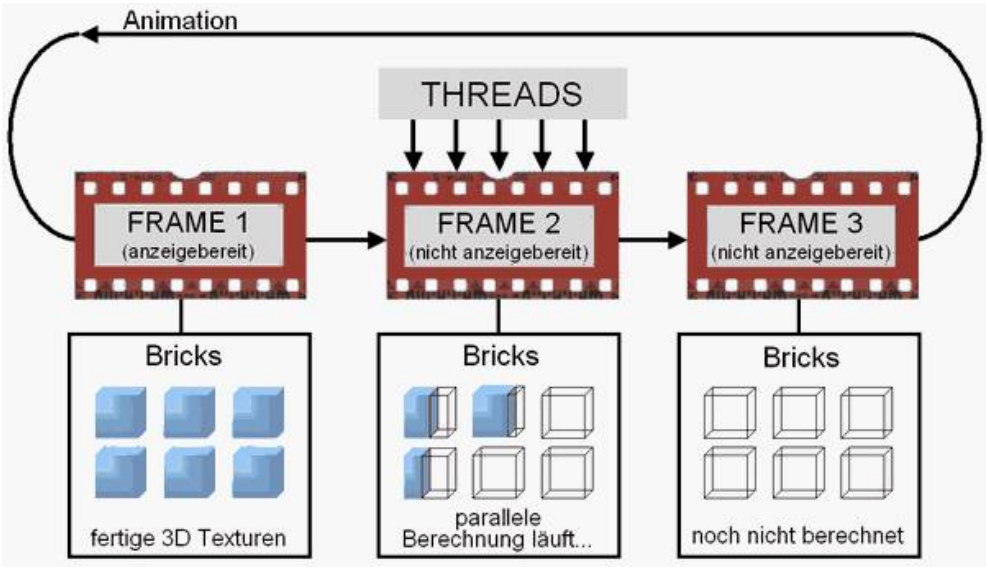



***Abbildung 7.49:*** *Die Frames eines Datensatzes werden sequentiell berechnet. Nur wenn alle Bricks eines Frames vollständig berechnet worden sind, kann der Frame dargestellt werden.*

**Ebene 2: Bricks**

Durch das Bricking des sondierten Datenbereichs erhält jeder Brick seinen eigenen Teilbereich des betroffenen Datenfensters. Je nach Anzahl der verfügbaren Threads (*threadcount*), der Anzahl der Bricks (*brickcount*) und der Texturauflösung der Bricks, wird die Gesamtmenge der zu berechnenden Daten des Datenbereichs durch Raumaufteilung auf unabhängig arbeitende Threads verteilt. Die Strategie wird durch Fallunterscheidung wie folgt definiert:

**Fall A: threadcount $\leq$ brickcount:** Jeder Thread muss eine oder mehrere vollständige 3D Texturen füllen. Hierzu wird die Anzahl der Bricks pro Thread durch

$\lfloor \text{brickcount}/\text{threadcount} + 0.5 \rfloor$ bestimmt. Durch die Rundung hat jeder Thread die gleiche Anzahl Texturen zu füllen. Der letzte Thread schafft durch mehr oder weniger Texturen den Ausgleich.

**Fall B: threadcount > brickcount:** Hier wird zusätzlich der zu füllende Speicher einer 3D Textur aufgeteilt. Die Zuordnung bestimmt sich aus der Anzahl der Threads pro Brick durch $\lfloor \text{threadcount}/\text{brickcount} \rfloor$ mit Rest $R$. Die übrigen Threads werden auf die ersten $R$ Texturen aufgeteilt.

Eine 3D Textur, welche von mehreren Threads bearbeitet wird, wird nun weiter in Flächen aufgeteilt, wobei die Richtung der Unterteilung durch die Achse mit der größten Auflösung festgelegt wird. Die Gesamtzahl der Flächen des Bricks wird auf die Threads verteilt, die diesem Brick zugewiesen sind.

Abbildung 7.50 veranschaulicht beide Fälle an einem Beispiel.

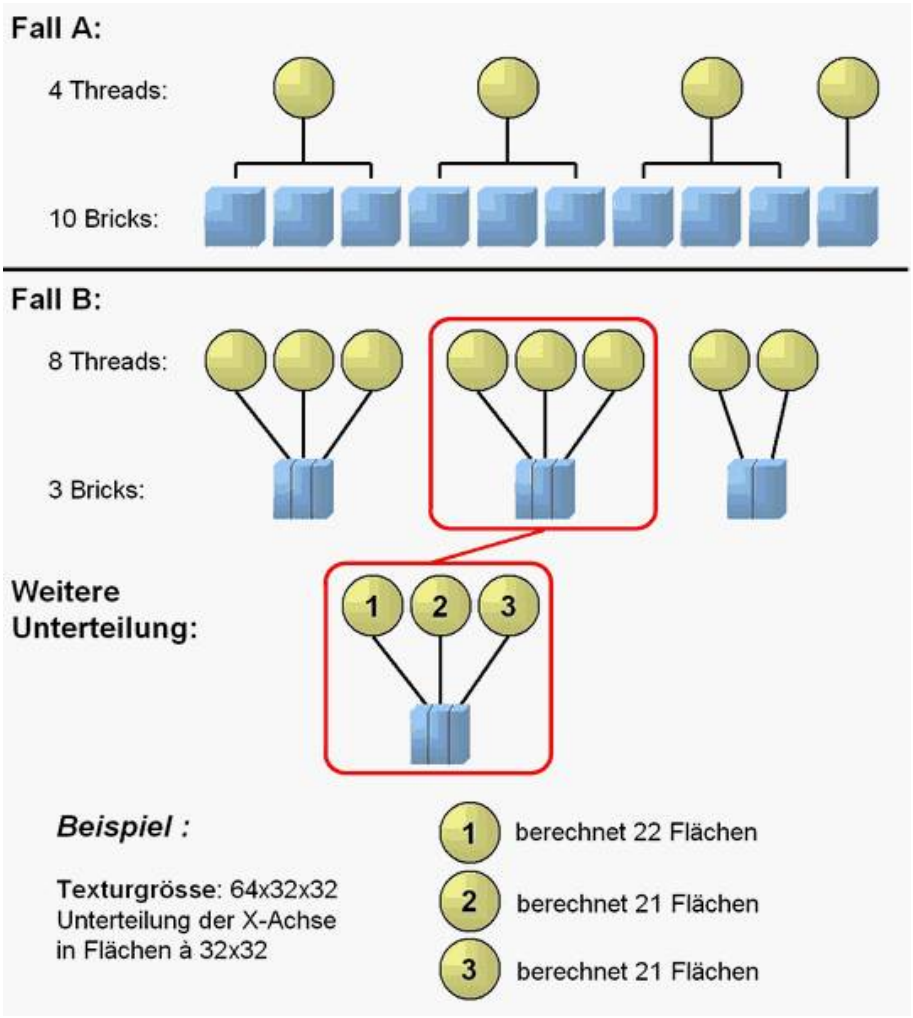



***Abbildung 7.50:*** *Beispiele für beide Fälle der Zuteilungsstrategie von Bricks und Threads*

#### 7.1.5.10 Klassifikation

Die Klassifikation unterscheidet sich, je nachdem, ob ein Volumen Rendering oder eine Iso-Flächen Extraktion geschieht.

**DirectVolume Rendering**

Die Klassifikation des Skalarfeldes durch die Anwendung beliebiger nicht-linearer Transferfunktionen, d.h. die Segmentierung der Skalarwerte eines Sehstrahlsegments, kann hardwarebeschleunigt realisiert werden. Die hierfür nötigen Gleichungen 7.19 und 7.20 werden für jede mögliche Kombinationen von $s_f$ und $s_b$ im quantisierten Intervall [0.0,. . . ,1.0] diskretisiert vorberechnet. Das Ergebnis dieser Vorberechnung ist ein zweidimensionales Datenfeld mit vier Komponenten pro Eintrag ($R, G, B$ aus Gleichung 7.20 und $\alpha$ aus Gleichung 7.19), das als 2D RGB$\alpha$ Look-Up Textur gespeichert wird. Durch die Anwendung der Dependent Texture-Fetch Technik kann die Klassifikation dann auf bequeme Weise hardwarebeschleunigt realisiert werden. Die 2D Texturkoordinaten entsprechen den Skalarwerten auf der Vorder- und der Rückseite ($s_f$ und $s_b$) eines Slabs. Jeder Farbwert in der vorberechneten 2D Textur mit den Koordinaten ($s_f$,$s_b$) entspricht der innerhalb eines Slabs kommulierten Farbe der linear interpolierten Skalarwerte (von $s_f$ nach $s_b$) am Anfang und Ende des Sehstrahlsegments, das durch die beiden Seiten des Slabs aus dem Sehstrahl herausgetrennt wird. Die Eingabe für die Vorberechnung sind vier nicht lineare Funktionen, jeweils für Rot, Grün, Blau und Transparenz, sowie ein Wert für die Dicke des Slabs. Abbildung 7.51 zeigt zwei Transferfunktionen mit den zugehörigen 2D Look-Up Texturen.

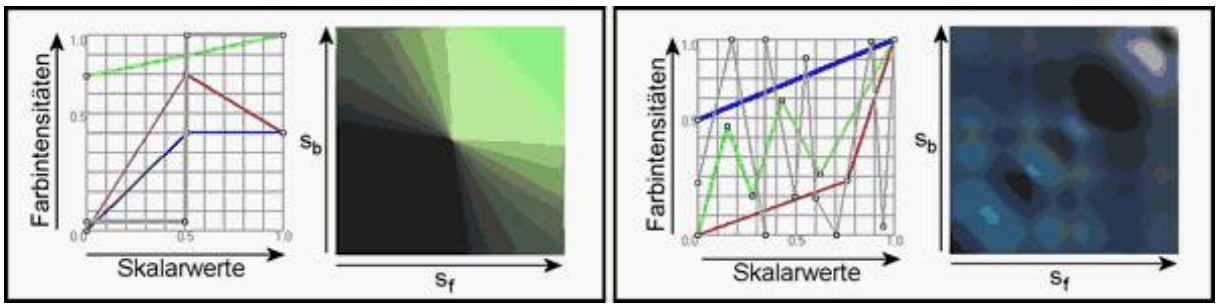



***Abbildung 7.51:*** *Zwei Beispiele für beliebige Transferfunktionen mit entsprechenden vor-integrierten 2D Look-Up Texturen. Die Transferfunktion beschreibt die Zuordnung Skalarwert → Intensität der Farbkanäle RGB$\alpha$. Die Texturkoordinaten der vor-integrierten 2D Textur sind die Skalarwerte auf der Vorder- und Rückseite eines Slabs ($s_f, s_b$). Die semi-transparente Look-Up Textur wurde hier zur Darstellung auf einen schwarzen Hintergrund gelegt.*

**Iso-Flächen**

Um auch bei dieser Visualisierungsmethode die Hardwarebeschleunigung des dependent texture fetchs auszunutzen, wird ebenfalls eine 2D Look-Up Textur erstellt. Der Inhalt dieser Textur basiert jedoch nicht auf einer mathematischen Formel, sondern wird unter Verwendung der vom Anwender erstellten Liste der Isowerte aufgebaut. Jeder Eintrag dieser Liste besteht aus einem Isowert ($s_{iso}$), der aus dem Skalarfeld als

Fläche extrahiert werden soll, und zwei Farben $(c_1, c_2)$, um beide Seiten der Fläche farblich unterscheiden zu können. Es muss nun für jeden Slab und jeden Sehstrahl festgestellt werden, ob sich auf dem eingeschlossenen Sehstrahlsegment ein Isowert befindet (siehe Abb. 7.52). Um die farbliche Unterscheidung beider Seiten der Isofläche durchführen zu können, nimmt man an, dass die Skalarfeldänderung zwischen Vorder- und Rückseite eines Slabs linear verläuft.

Die Betrachtung eines einzelnen Strahlsegments innerhalb eines Slabs liefert 3 Fälle:

**Fall A:** Die Skalarwerte am Anfang $(s_f)$, sowie am Ende des Strahlsegments $(s_b)$ liegen entweder beide oberhalb oder beide unterhalb des Isowertes $(s_{iso})$. Die lineare Verbindung schneidet den Isowert nicht, so dass die resultierende Farbe vollständig transparent ist $(\alpha = 0)$. Es gilt also der Fall $s_{iso} < s_f$, $s_{iso} < s_b$ oder $s_{iso} > s_f$, $s_{iso} > s_b$.

**Fall B:** Auf dem Sehstrahlsegment befindet sich ein Isowert, und zwar so, dass $s_f > s_{iso} > s_b$ gilt. Bei linearer Interpolation fallen die Skalarwerte von vorne nach hinten ab und schneiden dabei $s_{iso}$. Die Ausgabe ist daher nicht transparent $(\alpha > 0)$, weshalb $c_1$ als Farbe des Strahlsegments verwendet wird.

**Fall C:** Der Fall $s_f < s_{iso} < s_b$ umschliesst ebenfalls den Isowert, wobei hier die Skalarwerte von vorne nach hinten anwachsen. Nun verwendet man den zweiten vom Anwender definierten Farbwert $(c_2)$.

Eine korrekte Verdeckungsrechnung der Flächen wird durch back-to-front Rendering erreicht. Weiterhin ist es möglich, beliebig viele Isowerte zu definieren. Das hat jedoch aufgrund des verwendeten Algorithmus keine Auswirkung auf die Laufzeit, d.h. die Geschwindigkeit des Algorithmus ist unabhängig von der Anzahl der darzustellenden Isoflächen!

Abbildung 7.52 skizziert alle drei möglichen Fälle anhand von zwei Beispielen.

#### 7.1.5.11 Rendering

Die Durchführung des Slab-by-Slab Renderings mit einer 3D Textur erfordert die Berechnung der drei-dimensionalen Texturkoordinaten für die Vorder- und Rückseite des Slabs. Jeder Brick entspricht dabei einer 3D Textur, die den Teilbereich der Daten des sondierten Skalarfeldes als Texturdaten beinhaltet. Texturkoordinaten sind definitionsgemäss auf den Raum $[0, \ldots, 1]^3$ normiert. Durch eine Transformation $T$ der Koordinaten der Knotenpunkte eines Bricks in das Koordinatensystem einer 3D Textur werden die Kanten dieses Bricks auf die Kanten der 3D Textur abgebildet. Die Schnittpunkte der viewport-aligned Slices mit den Kanten eines Bricks werden durch die Transformation $T$ gleichermassen in den Texturraum abgebildet. Die Anwendung von $T$ auf die Koordinaten beider Flächen eines Slabs bildet dann auch den Slab im Koordinatensystem der 3D Textur ab. Das Textur-Sampling liefert die Skalarwerte $s_f$ und $s_b$ auf Pixel-Basis, welche auf der Vorder- und Rückseite des Slabs liegen.

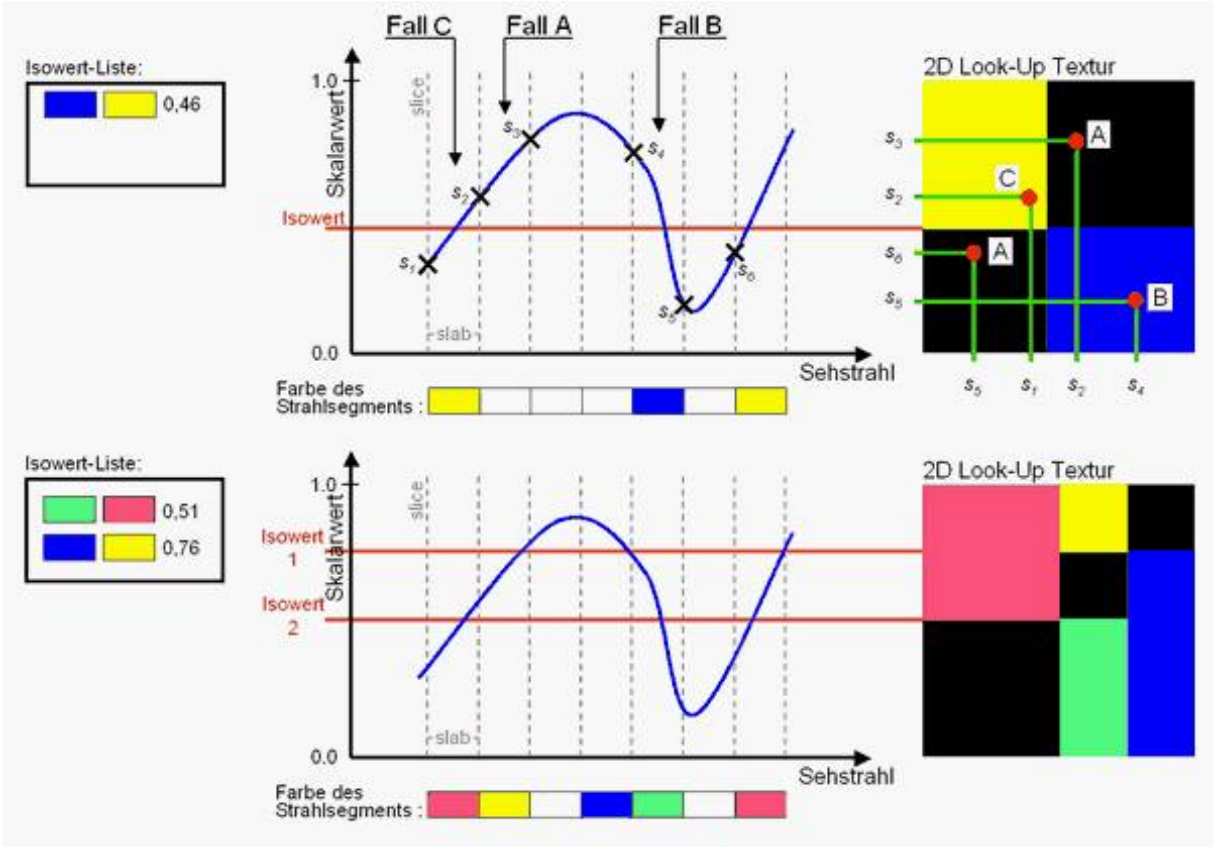



***Abbildung 7.52:*** *Zwei Beispiele für 2D Look-Up Texturen bei Isoflächen Rendering und die durch den dependent texture fetch ermittelten Farben der Sehstrahlsegmente. Im oberen Beispiel sind die drei möglichen Fälle eingezeichnet.*

Bei orthographischer Projektion des Volumens ist die Berechnung der Texturkoordinaten mit Hilfe der Transformation $T$ korrekt, da alle Sehstrahlen senkrecht zur Bildebene stehen (siehe Abbildung 7.53a).

Die perspektivische Projektion verzerrt jedoch das Erscheinungsbild der Flächen. Um ein Sehstrahlsegment innerhalb der 3D Textur trotzdem richtig weiterzuführen, müssen die Texturkoordinaten einer der beiden Flächen skaliert werden (siehe Abbildung 7.53b).

Da das Koordinatensystem der 3D Textur der perspektivischen Verzerrung durch OpenGL nicht automatisch angepasst wird, müssen die Texturkoordinaten der Rückseite des Slabs grösser skaliert werden, um der Perspektive „entgegenzuwirken“. Das Sampling der 3D Textur mit den Texturkoordinaten der Vorderseite, sowie den skalierten Texturkoordinaten der Rückseite liefert den korrekten Strahlverlauf der Sehstrahlsegmente durch das Skalarfeld innerhalb der 3D Textur. Man bezeichnet diese Skalierung als *Frontslice-to-Backslice Projection*, welche auch in diesem Konzept verwendet wird. Werden die Koordinaten der Rückseite des Slabs als Bezugskoordinaten verwendet, müssen die Texturkoordinaten der Vorderseite kleiner skaliert werden. Diese Projektion wird *Backslice-to-Frontslice Projection* genannt (siehe hierzu auch [EKE01]).

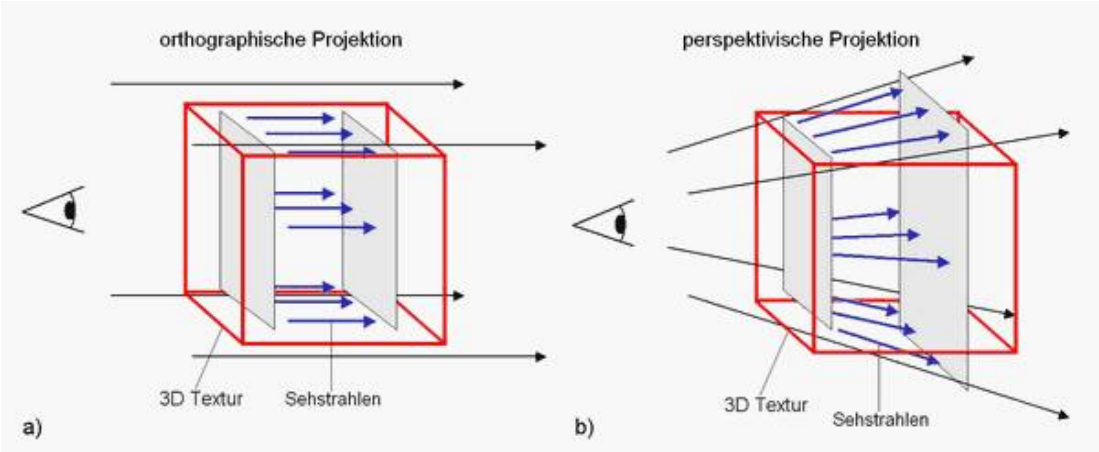



***Abbildung 7.53:*** *Der Verlauf der Sehstrahlen bei orthographischer (a) und perspektivischer (b) Projektion der Szene. Das Koordinatensystem der 3D Textur bleibt fest.*

Abbildung 7.54a zeigt die Lage zweier viewport-aligned Slices im Koordinatensystem einer 3D Textur. Der grau-schattierte Bereich auf der Rückseite des Slabs zeigt die nicht skalierte Projektion. Den Fehler, der entsteht wenn keine Skalierung der Texturkoordinaten bei perspektivischer Projektion durchgeführt wird, sieht man in Abbildung 7.54b.

Zur Berechnung der Koordinaten der viewport-aligned slices, welche die einzigen OpenGL-Pimitive darstellen, die hier benötigt werden, wird folgender Ansatz verwendet.

Der Quader (Bounding Box), der einen Brick der Sonde enthält, wird an jeder Kante auf Schnittpunkte mit der zur Bildebene parallel liegenden Fläche $F_d$ untersucht. Hierzu werden folgende Bezeichnungen der Knotenpunkte und Kanten gewählt:

Die mathematische Beschreibung der Kanten $G$ der Bounding Box mit Hilfe der Knotenpunkte $E$ lautet:

$$\begin{aligned} G_0(r) &= E_0 + r(E_1 - E_0) \\ G_1(r) &= E_1 + r(E_2 - E_1) \\ G_2(r) &= E_2 + r(E_3 - E_2) \\ &\text{usw} \ldots \end{aligned}$$

mit Parameter $r \in [0.0, \ldots, 1.0]$.

Die Orientierung der Bounding Box im Raum wird durch die OpenGL Modelview-Matrix $M$ festgelegt (siehe [WJS99]). Somit wird die Strecke $G'_i$ im Raum durch

$$G'_i(r) = M \cdot G_i(r)$$

beschrieben. Mit Hilfe der Parameterisierung durch $r$ können alle Punkte $\mathbf{X}$ auf der Strecke zwischen den Knoten $E_j$ und $E_k$ durch

$$\begin{pmatrix} m_{00} & m_{01} & \cdots \\ m_{10} & m_{11} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \cdot \left[ \begin{pmatrix} e_{j,x} \\ e_{j,y} \\ e_{j,z} \end{pmatrix} + r \cdot \begin{pmatrix} e_{k,x} - e_{j,x} \\ e_{k,y} - e_{j,y} \\ e_{k,z} - e_{j,z} \end{pmatrix} \right] = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \mathbf{X} \tag{7.24}$$

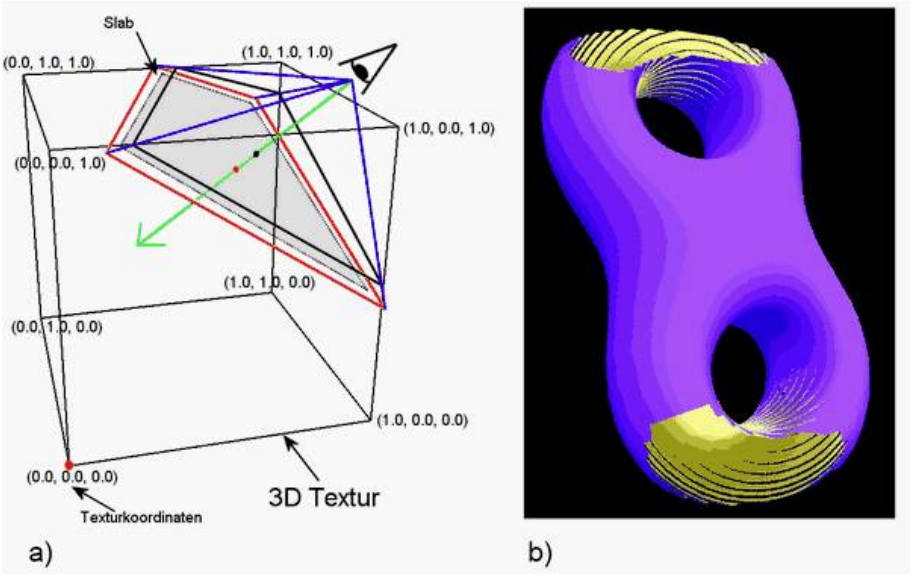



***Abbildung 7.54:*** *Perspektivische Projektion der Flächen innerhalb der 3D Textur (a). Die grau schattierte Fläche zeigt die orthographische (nicht-korrigierte) Projektion der Vorderseite des Slabs. Die rot umrahmte Fläche entspricht der korrigierten Rückseite. b) veranschaulicht den Fehler in der Darstellung, wenn die perspektivische Korrektur nicht durchgeführt wird.*

erfasst werden. Werden die Skalierungen und die Translation der Bounding Box durch die Modelview-Matrix $M$ eingebracht, stellen die Knotenpunkte einen Würfel mit der Kantenlänge 1 dar. Dann gilt $E_0 = (0, 0, 0)$, $E_1 = (1, 0, 0)$, usw...

Um nun die Schnittpunkte des Würfels mit der zur Bildebene parallelen Fläche $F_d$, welche sich auf der z-Achse an Position $d$ befindet, zu finden, muss festgestellt werden, ob die z-Koordinate des Punktes $\mathbf{X}$ auf dieser Fläche liegt. Hierzu wird die Berechnung der z-Koordinate in Gleichung 7.24 nach $r$ aufgelöst, und $z = d$ gesetzt. Der so erhaltene Wert für $r$ liegt im Intervall $[0.0, \ldots, 1.0]$, wenn die Strecke die Fläche $F_d$ schneidet. Ist dies der Fall, kann mit dem ermittelten Wert für $r$ und mit Gleichung 7.24 die x- und y-Koordinate des Schnittpunktes berechnet werden. Der gefundene Parameter $r$ liefert gleichzeitig eine Komponente der entsprechenden Texturkoordinate für die 3D Textur. Je nach untersuchter Kante wird $r$ oder $1-r$ verwendet. Die beiden anderen Komponenten sind 0 oder 1, ebenfalls abhängig von der untersuchten Kante. Siehe hierzu auch Abbildung 7.54a.

Mit diesem Ansatz wird nun der Algorithmus entworfen, um die Knotenpunkte einer *Viewport-Aligned Slice* an der Stelle $d$ der z-Achse zu berechnen.

Für jede Kante wird zunächst festgelegt, welche Verbindungen zu anderen Schnittpunkten erlaubt sind. Dies ist nötig, da sonst die Verbindungen der gefundenen Schnittpunkte durch den Würfel hindurch gehen können, und dadurch keine Fläche mehr erzeugen. Für jede Fläche sind sechs Kanten für eine weitere Schnittpunktsuche in Betracht zu ziehen. Durch die eingeführten Kantenbezeichnungen in Abb. 7.55 lassen sich die gültigen Nachbarkanten auflisten:

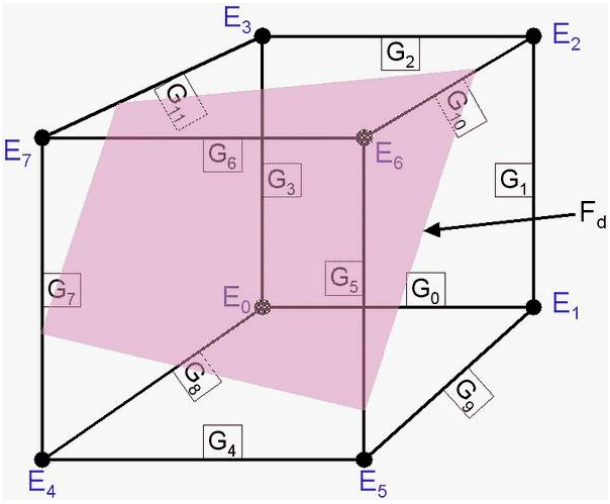



***Abbildung 7.55:*** *Die Bezeichnungen der Kanten und Knotenpunkte der Bounding Box eines Bricks.*

Kante 0 : 1, 2, 3, 4, 8, 9
Kante 1 : 0, 2, 3, 5, 9, 10
Kante 2 : 0, 1, 3, 6, 10,11
usw. . .

Desweiteren exisitert für jede Kante $G_i$ eine Markierung $m_i$, die speichert, ob Kante $G_i$ schon auf Schnittpunkte untersucht worden ist. Der Algorithmus lässt sich nun wie folgt formulieren:

- Suche den ersten Schnittpunkt der Fläche $F_d$ mit einer der zwölf Kanten des Würfels.
- Wurde ein Schnittpunkt mit Kante $G_i$ gefunden, dann markiere mit $m_i$, sonst Ende.
- Untersuche alle noch nicht markierten Kanten der Nachbarschaft von Kante $G_i$ auf Schnittpunkte mit der Fläche $F_d$. Die Suche wird abgebrochen, sobald die erste noch nicht markierte Kante $G_j$ gefunden wurde. Sind alle Nachbarkanten schon markiert, dann Ende.
- Markiere $m_j$ und wiederhole den letzten Schritt mit der gefunden Kante $G_j$.

Um einen Brick vollständig zu rendern, werden mit diesem Algorithmus im Abstand *slicedistance* alle Slices gezeichnet, welche durch die Bounding Box des Bricks geschnitten werden. Die slicedistance ist gleichzeitig die Dicke der Slabs, die das Skalarfeld scheibchenweise darstellen. Mit grösseren Abständen kann das Rendering zu Lasten der Bildqualität beschleunigt werden.

Für gleichmässige Übergänge zwischen den Bricks müssen die Flächen aller Bricks zusätzlich im gleichen Raster auf der z-Achse (Sichtachse) liegen. D.h. die erste Schnittebene, die bei jedem Rendering gefunden wird, bestimmt die Ausrichtung aller anderen Flächen der Bricks auf der durch *slicedistance* quantisierten z-Achse.

#### 7.1.5.12 Shader Implementierung

Die Shaderimplementierung zerfällt in zwei Bereiche, die hier nur kurz vorgestellt werden.

**Vertex Shader**

Hier werden alle vektoriellen Berechnungen durchgeführt, welche aufgrund einer neuen Orientierung des Volumens im Raum erforderlich sind. Hierzu gehört die Durchführung der *Frontslice-to-Backslice* Projection, d.h. das Ermitteln der skalierten Texturkoordinaten der Rückseite, welche sich mit Hilfe der Transformation $T'$ angewendet auf die Knotenkoordinaten der Vorderseite berechnen lässt. $T'$ entspricht der Transformation $T$ mit einer zusätzlichen Skalierung um die perspektivische Projektion auf Basis des Koordinatensystems der 3D Textur durchzuführen. Weiterhin werden Vektoren, welche für die Beleuchtungsrechnung benötigt werden, vorbereitet.

Es werden folgende Eingaben im Vertex Shader für das Rendering einer Slice, bzw. eines Slabs verarbeitet:

- Koordinaten der Knotenpunkte der Slice,
- Orientierung des Volumens (als Matrix),
- Distanz der Kamera und Dicke eines Slabs, sowie
- Positionen der Lichtquellen für diffuses und spekulares Licht.

Die Ausgabe des Shaders pro Durchlauf ist:

- Transformierte Koordinaten des Knotens,
- zwei Texturkoordinaten (Slab im Koordinatensystem der 3D Textur) und
- bei Beleuchtung den normalisierten Lichtvektor, sowie den normalisierten $\vec{H}$ Vektor für den spekularen Anteil im Phong-Modell.

**Fragment Shader / Program**

Auf der Pipelinestufe des Fragment Shaders wird der *Dependent-Texture Fetch* durchgeführt. Für das Rendering eines Slices werden deren Knotenpunkte und die entsprechenden Texturkoordinaten des Slabs im Koordinatensystem der 3D Textur benötigt. Die Aktionen des Fragment Shaders auf per-Fragment Basis gliedern sich in zwei Pässe: *Address-Shader* und *Color-Shader*. Im ersten Pass müssen die Texturkoordinaten für die 2D Look-Up Textur ermittelt werden. Durch das Textur-Sampling unter Einbeziehung der rasterisierten Texturkoordinaten auf per-Fragment Basis können die Skalarwerte $s_f$ und $s_b$ der Vorder- und der Rückseite eines Slabs aus der 3D Textur tri-linear interpoliert gelesen werden. Der *Address-Shader Pass* erzeugt die Texturkoordinaten für die 2D Textur, indem der Skalarwert $s_f$ als x-Koordinate und der Skalarwert $s_b$ als y-Koordinate interpretiert wird. Im *Color-Shader Pass* findet dann das Sampling der 2D Look-Up Textur statt. Je nach Visualisierungmethode beinhaltet die 2D Textur Farbwerte der vorintegrierten Klassifikationen oder Farben der Isoflächen.

Die Durchführung der oben genannten Aktionen des Fragment Shaders mit den Informationen einer Fläche der *Viewport-Aligned Slices* liefert das klassifizierte Skalarfeld,

welches sich innerhalb des zugehörigen Slabs befindet. Ein *Back-to-Front Rendering* aller am Volumen-Rechteck (bzw. am Brick) geclippten Flächen stellt den vollständigen Inhalt der 3D Textur klassifiziert und unschattiert dar.

Zur Simulation einer Beleuchtung des Volumens müssen die Normal-Vektoren auf Pixelbasis bekannt sein. Bei der Erstellung der 3D Texturen wurden die Gradienten, falls eine Beleuchtungsrechnung durchgeführt werden soll, berechnet und komponentenweise pro Texel in den Farbkanälen $R, G, B$ abgelegt. Das Textursampling vor dem *Address-Shader Pass* liefert dann nicht nur die beiden Skalarwerte der Vorder- und Rückseite des Slabs, sondern ebenfalls die Gradienten $\vec{G}_{front}$ und $\vec{G}_{back}$ am Anfang und am Ende des Sehstrahlsegments.

Beim *DirectVolume Rendering* wird das semi-transparente Strahlsegment in einem Pixel zusammengefasst. Zur Beleuchtungsrechnung genügt es, den gemittelten Gradient $\vec{G}_{avg}$ aus $\vec{G}_{front}$ und $\vec{G}_{back}$ (siehe [EKE01]) oder sogar nur den Vektor $\vec{G}_{front}$ der vorderen Fläche zu verwenden.

Beim Schattieren von Iso-Flächen ist es für gleichmässige Farbübergänge auf der beleuchteten Fläche wichtig, dass der Gradient, der bei der Berechnung der Punktprodukte (vgl. Gleichung 3.2) verwendet wird, korrekt zwischen den verfügbaren Gradienten der Vorder- und Rückseite des Slabs interpoliert wird. Das hierzu nötige Interpolationsgewicht $IP$ berechnet sich durch $IP = (s_{iso} - s_f)/(s_b - s_f)$ für den Gradienten der Rückseite, und $1 - IP$ für den der Vorderseite (vgl. dazu Abb. 7.56 und [EKE01]). Der interpolierte Gradient ist demnach $\vec{G} = \vec{G}_{back} * IP + \vec{G}_{front} * (1 - IP)$. Zur linearen Interpolation von zwei Vektoren verfügt die `GL_ATI_fragment_shader` Extension über entsprechende Befehle (siehe Spezifikation in [opea]).

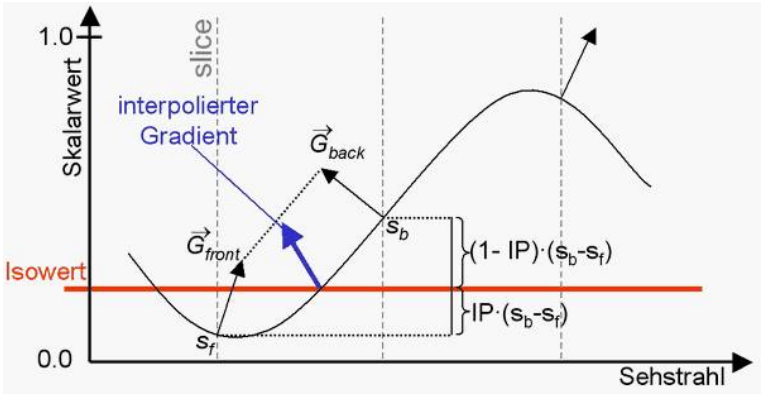



***Abbildung 7.56:*** *Der Gradient an einem Pixel der Iso-Fläche bestimmt sich durch lineare Interpolation der Gradienten am Anfang und am Ende des Sehstrahlsegments.*

Die Berechnung des Interpolationsgewichtes $IP$ für Isoflächen kann nicht direkt im Fragment Shader implementiert werden, da der Wert $s_{iso}$ des betroffenen Iso-Wertes nicht mehr verfügbar ist. Aus diesem Grund wird für die Visualisierung von Isoflächen eine zweite 2D Look-Up Textur (*IP-Textur*) vorberechnet, welche die Interpolationsgewichte speichert. Durch das *Dependent Texture Fetch* Verfahren kann $IP(s_f, s_b)$ demnach indirekt ermittelt werden.

Mit den ermittelten Gradienten können nun die Anteile des diffusen und des spekularen Lichts zur ambienten Farbe (aus der 2D Textur) nach Gleichung 3.2 hinzuaddiert werden.

Die Eingabedaten des Fragment Shaders sind:

- 3D Textur mit Materialdaten und (bei Beleuchtung) Gradienten
- 2D dependent texture mit vorintegrierter Klassifikation oder Isowert-Muster
- 2D dependent texture mit Interpolationsgewichten für die Gradienten (nur bei Isoflächen mit Shading)
- normalisierter Richtungs-Vektor des diffusen Lichts (nur bei Shading)
- normalisierter H-Vektor für den spekularen Anteil (nur bei Shading)
- Texturkoordinaten der Vorderseite des Slabs
- Texturkoordinaten der Rückseite des Slabs

Die Ausgabe des Fragment Shaders sind die Farben der Sehstrahlsegmente eines Slabs auf Pixel-Basis, mit denen die Vorderseite des Slabs in den Framebuffer gerendert wird. Bei Beleuchtung wurde die Farbe unter Verwendung des ermittelten Gradienten und der Einstellungen der diffusen und spekularen Lichtquelle entsprechend modifiziert.

Die Anzahl der verfügbaren Pässe bzw. Indirections ist entscheidend für die möglichen Berechnungen auf per-Fragment Basis. Unter Verwendung der Fragment Program Extension ist es möglich, die für die Beleuchtungsrechnung erforderlichen Gradienten jedes einzelnen Skalarwertes nun in der Fragment Program Pipelinestufe zu berechnen. Mit Hilfe mehrerer Textursampling-Pässe können die Nachbartexels eines Texels gelesen werden, um damit die Gradienten zur Laufzeit zu rekonstruieren, anstatt sie in zusätzlichen 3D Texturen von der CPU vorberechnet speichern zu müssen. D.h. für *DirectVolume* und *IsoSurfaces*, sowie für schattierte und nicht-schattierte Darstellung beinhalten die 3D Texturen der einzelnen Bricks dann ausschliesslich die Skalarwerte des sondierten Teilbereichs der darzustellenden Datenquelle.

Hierfür wird die Verwendung der Extension `GL_ATI_fragment_shader` durch ARB-Extension `GL_ARB_fragment_program` abgelöst. Für das Rendering von beleuchteten Volumen müssen im Fragment Program die Nachbartexels des zu schattierenden Skalarwertes gelesen werden, um die differentielle Änderung des Skalarfeldes an dieser Stelle zu bestimmen. Die Gleichungen für die Ausführung im Fragment Program werden wie folgt verwendet:

$$\begin{aligned} G_x(i,j,k) &= S(i+s_x,j,k) - S(i-s_x,j,k) \\ G_y(i,j,k) &= S(i,j+s_y,k) - S(i,j-s_y,k) \\ G_z(i,j,k) &= S(i,j,k+s_z) - S(i,j,k-s_z) \end{aligned} \tag{7.25}$$

mit $s_x$, $s_y$ und $s_z$ als Grösse eines Auflösungsschrittes der Textur in den auf das Intervall $[0.0,\ldots,1.0]$ normierten Texturkoordinaten. Die Division durch die Seitenlänge fällt weg, da der Gradient nachträglich normiert wird. Dies geschieht durch

$$G_{norm}(i,j,k) = \frac{G(i,j,k)}{|G(i,j,k)|}. \tag{7.26}$$

#### 7.1.5.13 Panels

Die Panels zu den beiden Visualisierungsmethoden zerfallen in einen Bereich, den beide Panels gemeinsam haben, und einen Bereich, der spezifisch für die jeweilige Methode ist.

**Gemeinsamer Teil**

Für beide Visualisierungsmethoden existieren gemeinsame Konfigurationsteile, welche aufgrund des Brickings und deren Aufteilung auf Threads in Abhängigkeit der eingestellten Texturauflösung zustande kommen. Weiterhin kann der Abstand der Flächen, sowie die Anzahl der zu verwendeten Threads zur Vorberechnung verändert werden. Der gemeinsame Bestandteil (*VolumeSettings*) der Panels beider Methoden erlaubt dem Benutzer, diese Einstellungen zu modifizieren (Abbildung 7.57).

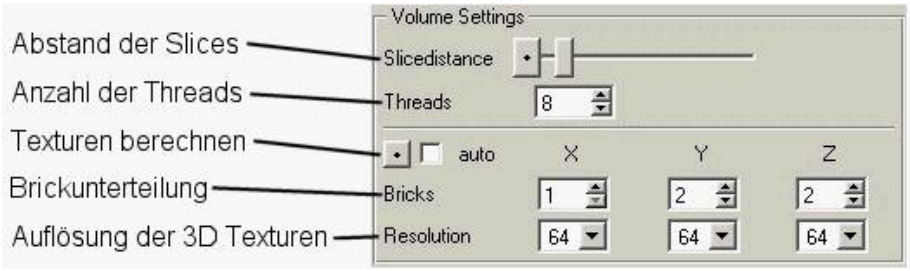



***Abbildung 7.57:*** *Die Gruppe der VolumeSettings Bedienelemente ermöglicht das Modifizieren von Einstellungen für texturbasierte Rendering-Methoden.*

**Direct-Volume Renderer**

Während der Visualisierung des Volumens mit DirectVolume kann der Anwender die vier Transferfunktionen in einer grafischen Ansicht modifizieren. Bei der Variante mit aktivierter Schattierung kann zusätzlich die Intensität der Schattierung verändert werden. Das VisModulePanel für DirectVolume zeigt Abbildung 7.58.

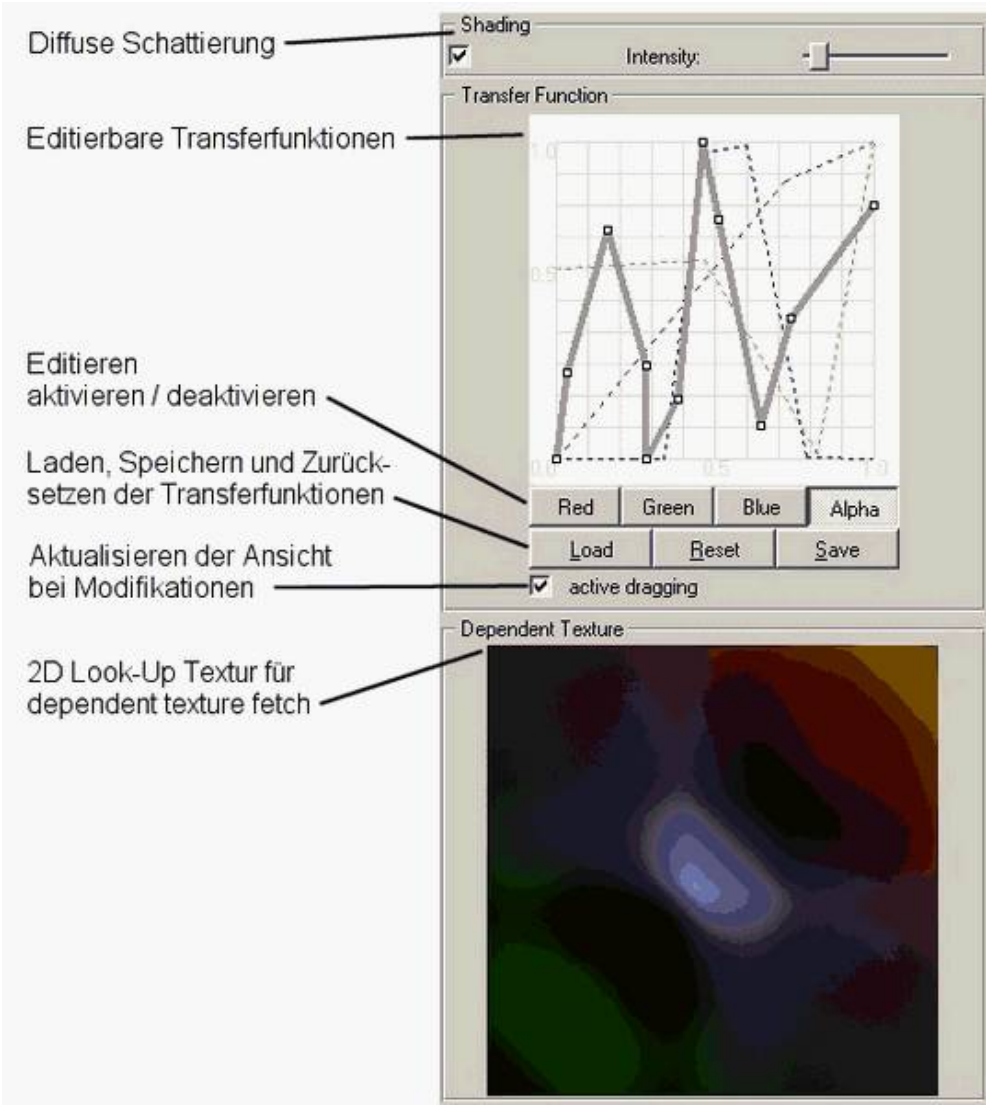



***Abbildung 7.58:*** *Das VisModulePanel des DirectVolume Renderers*

### Iso-Surface Renderer

Das VisModule Panel des IsoSurface-Renderes bietet dem Anwender die Möglichkeit, die Isoflächen in Form einer Liste festzulegen. Zudem können alle erwähnten Eigenschaften der diffusen und spekularen Lichtquelle verändert werden. Abbildung 7.59 zeigt das IsoSurface Panel.

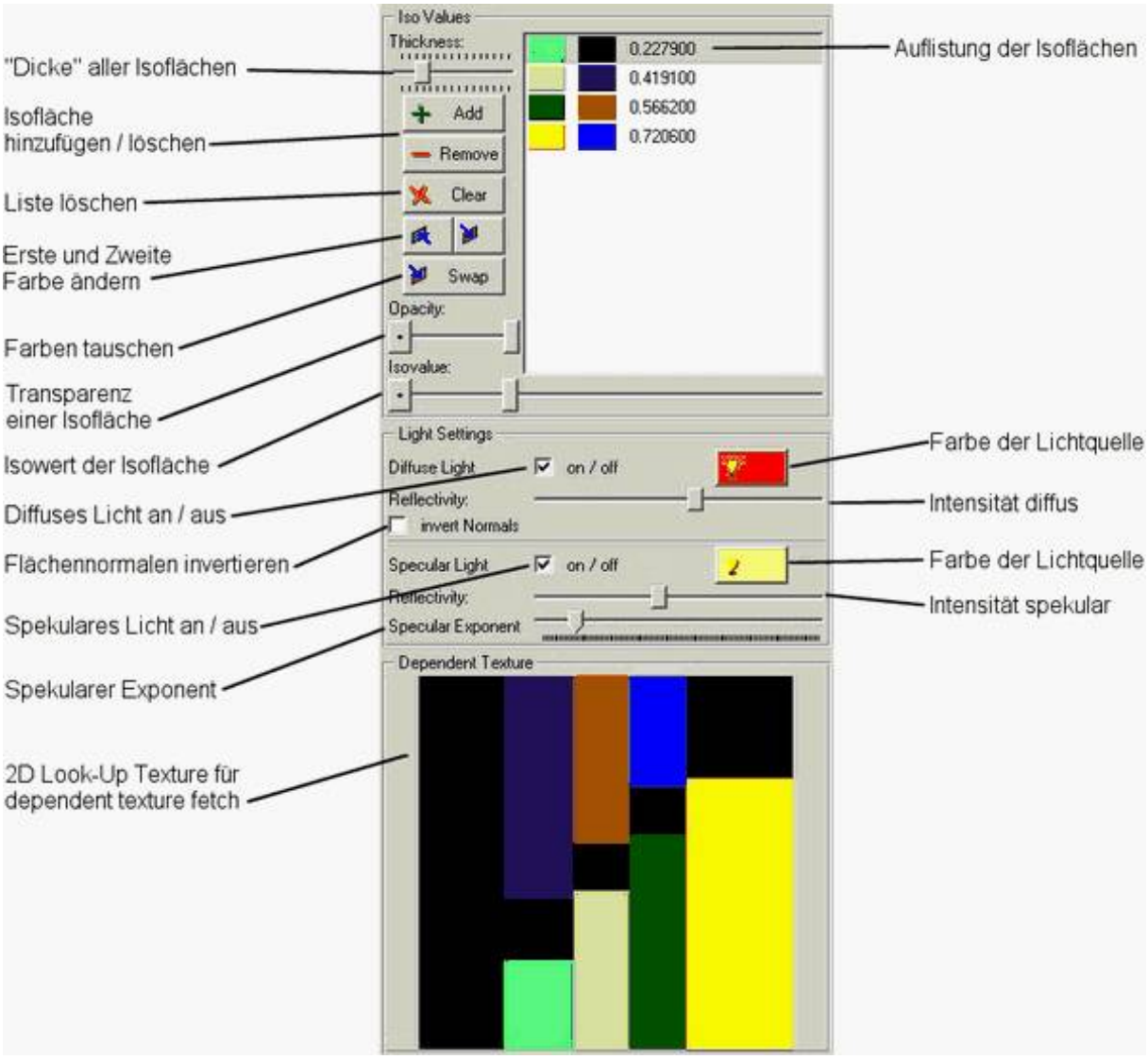



***Abbildung 7.59:*** *Das VisModulePanel des IsoSurface Renderers*

#### 7.1.5.14 Klassenstruktur

Der Klassenaufbau der beiden Visualisierungsmethoden ist ähnlich denen, der bereits vorgestellten Methoden. Auch hier wird wieder von den vorgegebenen virtuellen Mutterklassen geerbt und die zugehörigen Funktionen überladen. Eigene Actions werden definiert. Zusätzlich wird noch eine neue Basisklasse implementiert: `ViewAlignedBrick`. Sie kapseln das weiter oben beschriebene Bricking in eine eigene Klasse (Abbildung 7.60).

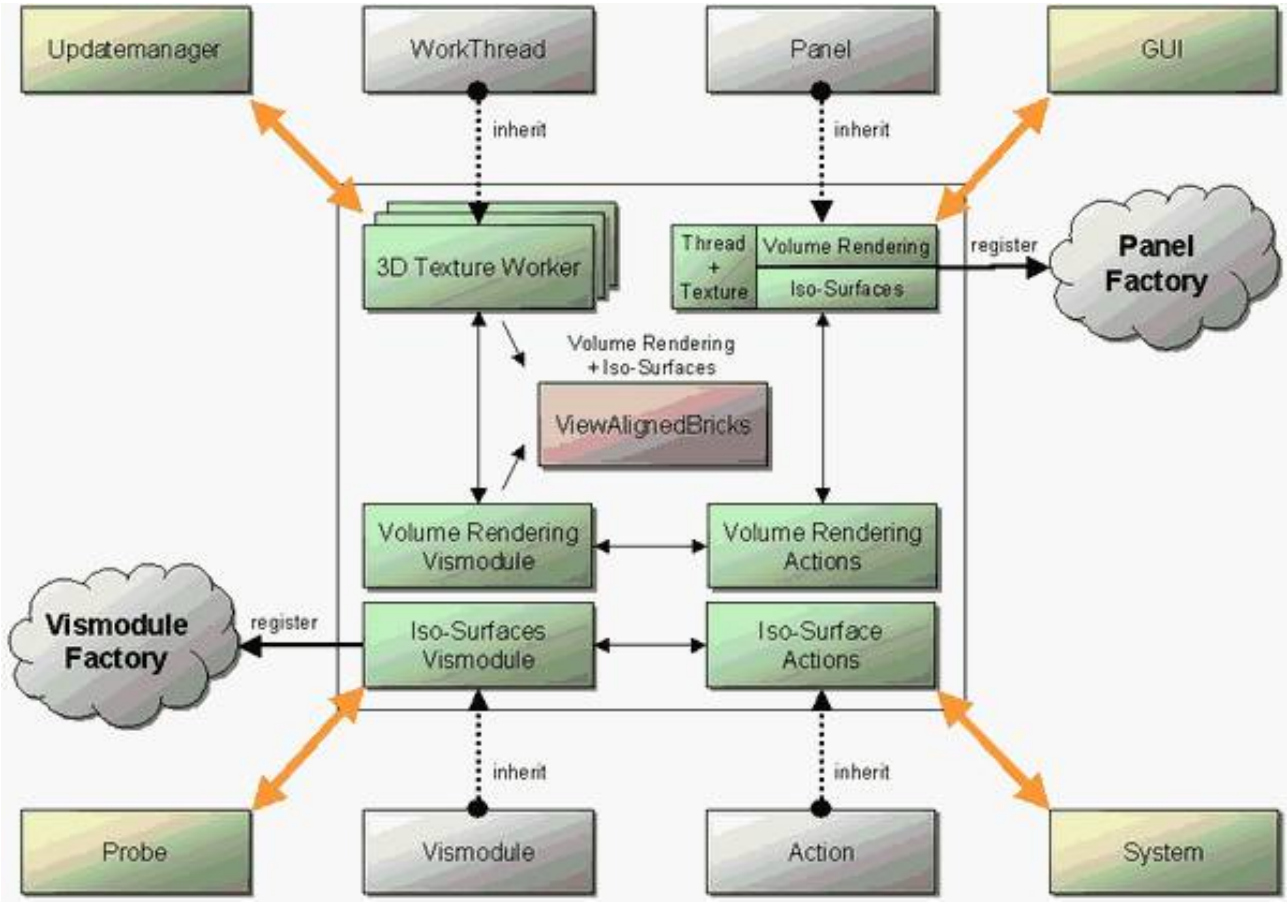



***Abbildung 7.60:*** *Die Klassen, die für die Implementierung des Volume Rendering / Iso-Surface Verfahrens verwendet werden.*

#### 7.1.5.15 Ergebnisbilder

In den folgenden 4 Abbildungen 7.61, 7.62, 7.63, 7.64, 7.65 und 7.66 werden einige weitere Ergebnisbilder der beiden vorgestellen 3D Textur basierten Visualierungsmethode gezeigt.

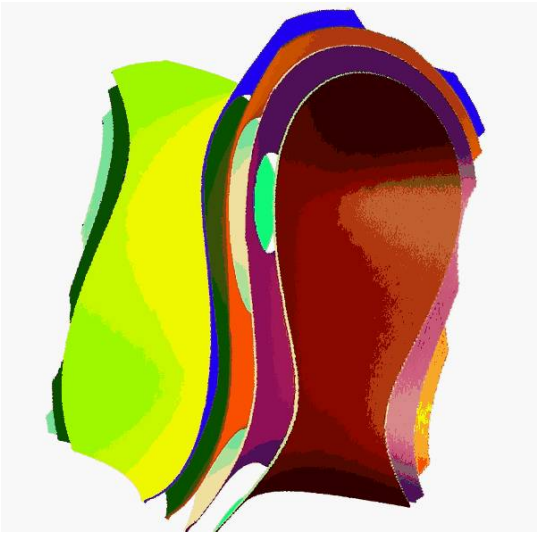


***Abbildung 7.61:*** *3D Texture Iso-Surface Beispiel: Mehrere Iso-Flächen, die aus einem mathematisch generierten (harmonischen) Datensatz extrahiert werden.*

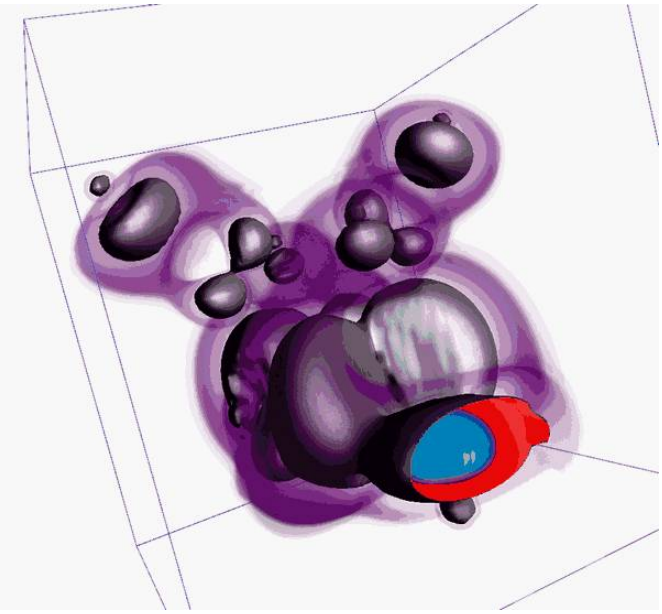


***Abbildung 7.62:*** *Gleichzeitiges Volumen und Iso-Surface Rendering eines medizinischen Datensatzes.*

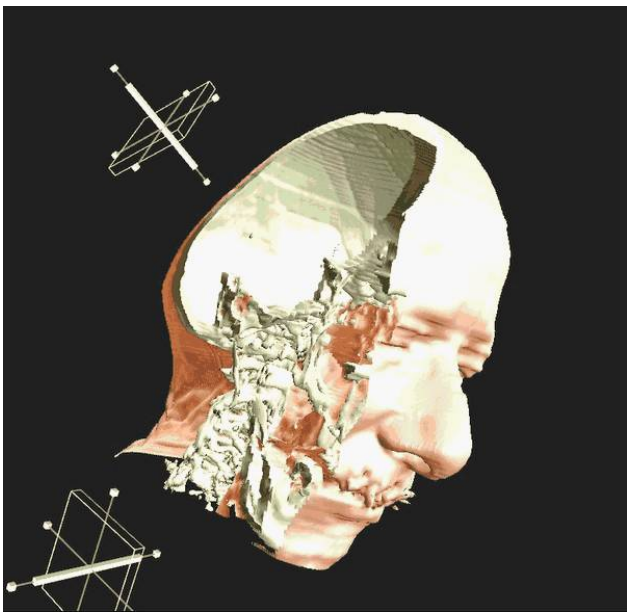


***Abbildung 7.63:*** *Darstellung eines medizinischen CT Scandatensatzes mit Hilfe von zwei Cut-Planes und Iso-Surfacces.*

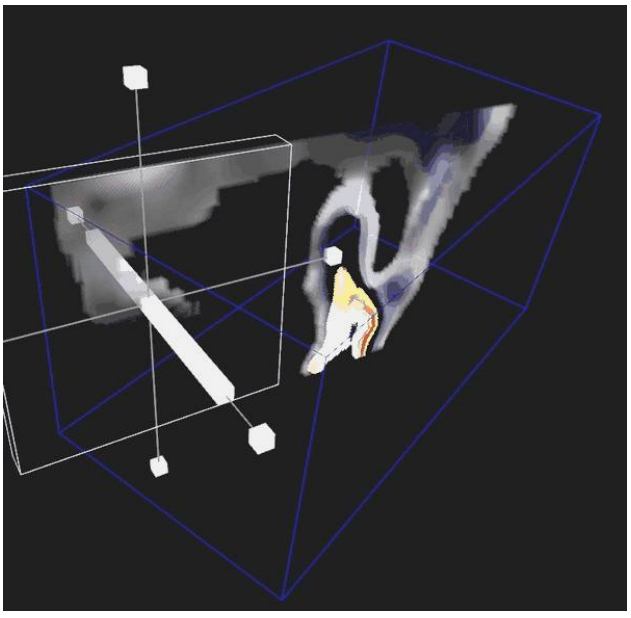


***Abbildung 7.64:*** *Iso-Flächen Darstellung eines Feuersimulations-Datensatzes mit aktiverter CuttingPlane.*

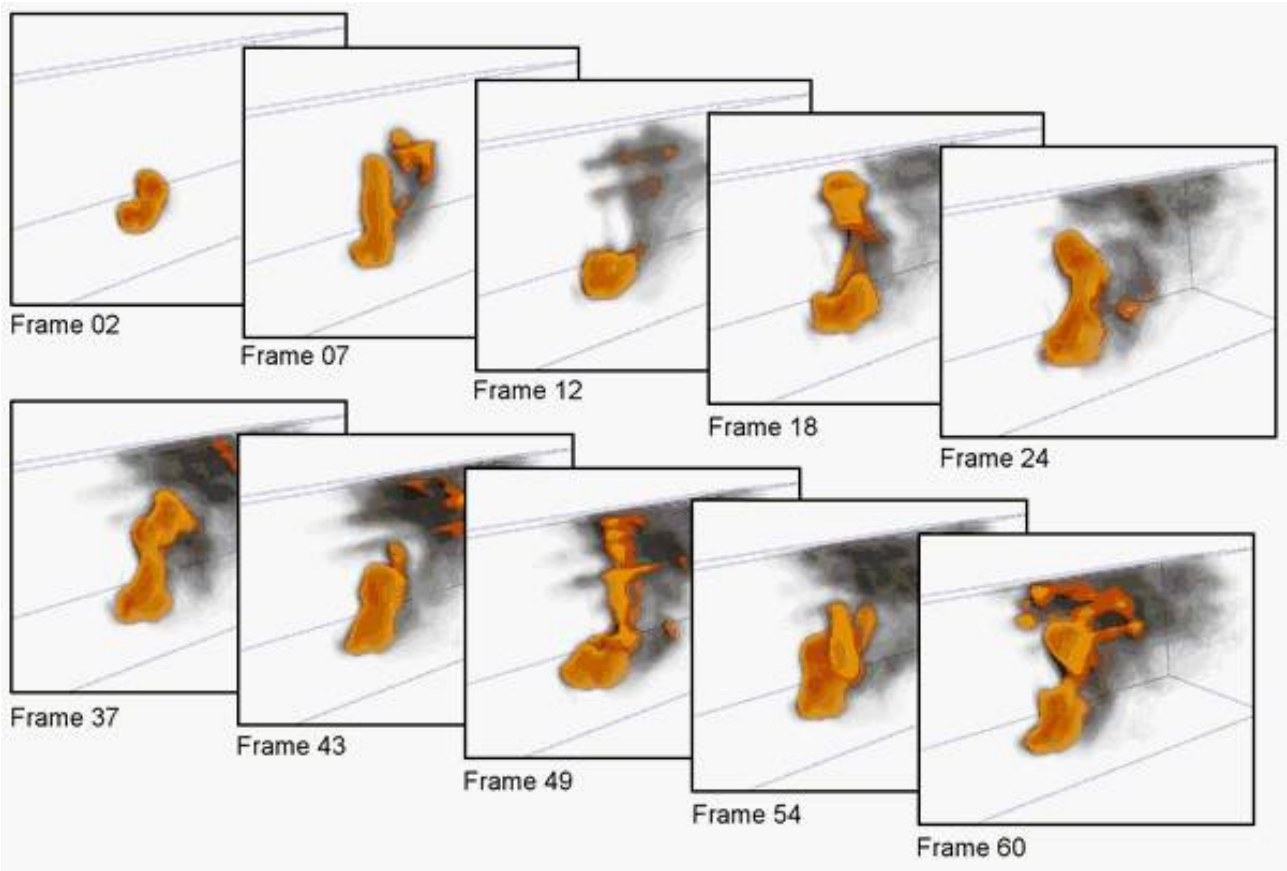



***Abbildung 7.65:*** *Auszug aus einer Animation eines Feuer Datensatzes (Volumen Rendering).*

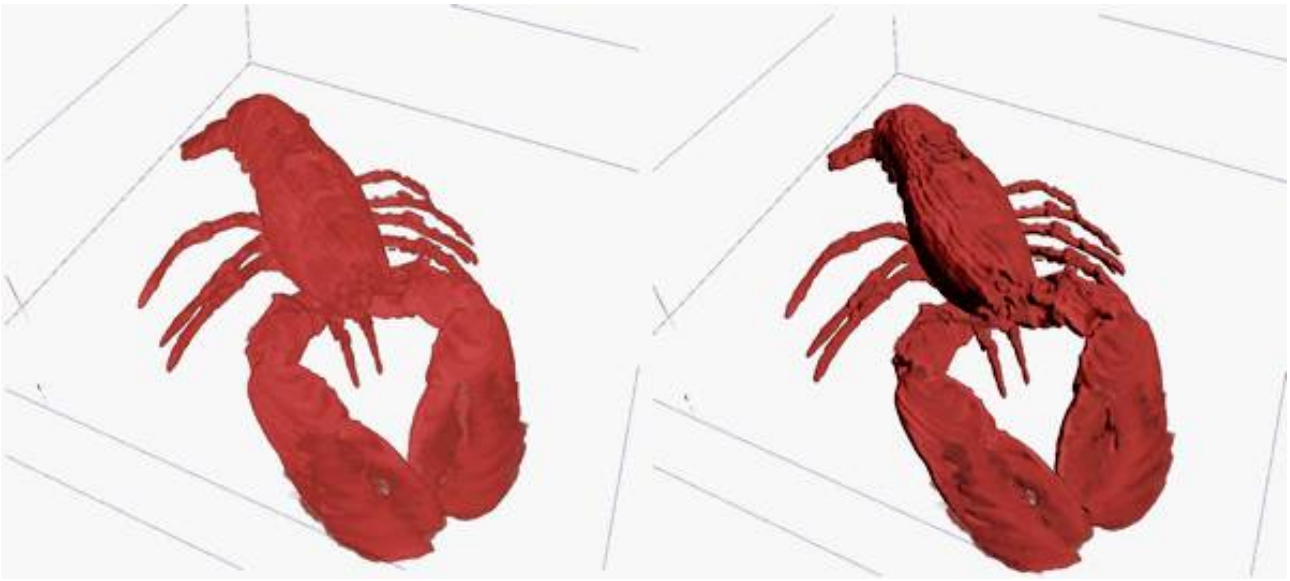


***Abbildung 7.66:*** *Direkter Vergleich zwischen schattiertem (shaded) und normalem Volumen Rendering mit Hilfe eines Röntgen Datensatzes eines Hummers).*

### 7.1.6 Übersicht

Neben den in Kapitel 7.1.1 bis Kapitel 7.1.5 ausführlicher vorgestellten Visualisierungsmethoden wurden im Rahmen der Implementierung noch eine Reihe weiterer Methoden entwickelt und in das System integriert. An dieser Stelle wird eine Übersicht gegeben, für welche Dimension des Datensatzes (Skalar, Vektor) welche Art von Visualisierungsmethoden insgesamt realisiert wurden, ohne auf die einzelnen Methoden weiter einzugehen (Tabelle 7.1 und 7.2, gefolgt von einigen exemplarisch ausgewählten Beispielen).

#### 7.1.6.1 Skalarfeld Visualisierungen

| **Methoden-Name** | **Parallelisierungsart** |
|---|---|
| Data Grid | Datenraum |
| Marching Cubes Iso-Surfaces | Datenraum |
| Marching Tedrahedron Iso-Surfaces | Datenraum |
| Volume Rendering | Datenraum |
| Fractal Volume Rendering | Datenraum |
| 3D Textur Iso-Surfaces | Datenraum |

***Tabelle 7.1:*** *Übersicht über die zur Verfügung stehenden Skalarfeld-Visualisierungsmethoden.*

**Data Grid**

Bei dieser Visualisierungsmethode, die in Kapitel 7.1.1 kurz vorgestellt wurde, wird ein regelmäßiges Gitter durch das Datenfeld gelegt. Hierbei ist die genaue Auflösung des Gitters in $x$, $y$ und $z$ Richtung getrennt regelbar. An die Gitterpunkte wird der vorgefundene Skalarwert als lesbarer Text platziert. Zusätzlich kann der Gitterpunkt noch mit einer justierbaren Farbverlaufskala anhand seines relativen Wertes im Datenfeld gefärbt werden (Abbildung 7.1).
Diese Visualisierungsmethode eignet sich in erster Linie für Debugging Zwecke und die schnelle Analyse des Datenfeldes. Peaks im Feld fallen sehr schnell anhand ihres Farbwertes auf. Durch die Ausgabe des Zahlenwertes können die Daten sehr genau, beispielsweise um kritische Stellen herum, abgelesen werden. So kann beispielsweise die Korrektheit der zugrundeliegenden Simulation als auch der Import / Transfer der Daten auf Konsistenz geprüft werden. Bei hoch gewählter Auflösung des Gitters wird die Darstellung schnell unübersichtlich.

**Marching Cubes Iso-Surfaces**

Marching Cubes (MC) repräsentiert das klassische Iso-Flächen Extraktionsverfahren. Auf Basis einer würfelförmigen Zerlegung des Datenraumes werden die Zellen gefunden, durch die die gesuchte Iso-Fläche verläuft. Anhand der Skalarwerte an den Eckpunkten der Fläche wird der Verlauf der Fläche durch die Zelle bestimmt. Die ermittelte Fläche entsteht durch das Zusammensetzen der so berechneten Teilflächen (Dreiecke). Je feiner die Würfelzerlegung gewählt wird, bzw. je genauer sie sich der Originalgitter Auflösung

annähert, um so exakter wird die berechnete Iso-Fläche [EaSK96] (Abbildung 7.67 (a)). Diese Visualisierungsmethode eignet sich, um z.B. in Wärmefeldern die Fläche mit konstanter Temperatur zu ermitteln oder in Druckfeldern die Fläche mit einheitlichem Luftdruck.

**Marching Tedrahedron Iso-Surfaces**

Das Marching Tedrahedron (MT) Verfahren ist eine Verfeinerung des MC Verfahrens. Anstatt den Raum gleichmäßig in Würfel zu zerlegen, wird jeder dieser Würfel zusätzlich in 5 Tedraeder zerteilt. Anhand der Skalarwerte an den Eckpunkten der Tedreader wird für jeden von ihnen der Verlauf der gesuchten Iso-Fläche bestimmt. Die ermittelten Dreiecksflächen werden zur Iso-Fläche kombiniert [EaSK96] (Abbildung 7.67 (b)). Der Vorteil gegenüber dem MC Verfahren liegt in erster Linie darin, dass durch die feinere und nicht achsenparallele Aufteilung des Datenraumes „glatter“ wirkende Flächen berechnet werden. Allerdings bringt die höhere Zellanzahl (Tedraeder anstatt Würfel) und deren Ausrichtung (nicht entlang der Raumachsen) auch einen höheren Rechenaufwand mit sich im Vergleich zu der auf Marching Cubes basierenden Variante.

**Volume Rendering**

Mit einer Transferfunktion produziert das Volumen Rendering „Röntgenbild“ ähnliche Bilder (vgl. Implementierung Kapitel 7.1.5). Die vorgefundenen Skalarwerte werden als „Dichte“ bzw. „Menge der Daten“ am ausgelesenen Raumpunkt interpretiert. Je „dichter“ die Daten, d.h. je größer der vorgefundene Skalarwert, desto stärker / heller wird der Punkt am Ende dargestellt. Dabei addieren sich die Dichtewerte über die räumliche Tiefe (Abbildung 7.69).

Mit dieser Visualisierungsmethode lassen sich schnell Datenbereiche mit hohem Datenaufkommen bzw. großer Dichte identifizieren. Auch lassen sich einfach Bereiche unterschiedlicher Dichte voneinander unterscheiden (z.B. ein „Loch“ in einem ansonsten gleichmäßig besetzten Volumen). Durch das Verändern der Transferfunktion lassen sich bestimmte Dichtebereiche aus der Datenmasse herausfiltern und verschieden einfärben. So kann man beispielsweise in einem Gasgemisch schnell Bereiche identifizieren, die kritische Konzentrationen giftiger Gase enthalten bzw. diese Bereiche klassifizieren. Durch zusätzliches Shading entsteht der räumliche Eindruck der dargestellten „Wolken“, da sich Licht und Schattenbereiche ausbilden.

**Fractal Volume Rendering**

Diese Erweiterung des Volumen Renderings beruht auf demselben Prinzip. Kurz vor der Ausgabe der Daten wird diesen eine definierbare lokale fraktale Verzerrung hinzugerechnet, die sich an der dort vorliegenden Strömung und deren Stärke orientiert. Es entsteht je nach Parameterwahl der Eindruck von kleinen „Schlieren“ oder auch „Flammenzungen“ (Abbildung 7.77).

Diese Visualisierngsmethode eignet sich in erster Linie für die realistisch wirkende Darstellung von Gasen bzw. Nebel oder Feuer. In der Regel sind die Auflösungen, die die Simulation zur Berechnung verwendet, zu „grob“, um bei der anschließenden Visualisierung einen „echt“ erscheinenden Eindruck beim Betrachter zu erwecken. Wird ein Zimmerbrand beispielsweise mit einer Gitterauflösung von 3x3cm berechnet, so steht in der anschließenden Darstellung des Feuers oder Qualms maximal diese Auflösung zur

Verfügung. Die Visualisierung hätte einen entsprechenden „blockartigen“ Charakter. Feuer hat allerdings in der Natur die Eigenschaft, kleine Flammenzungen auszubilden, genau wie Rauch bzw. Qualm die Eigenschaft hat, wie zerfranste „Watte“ zu wirken. Diese feinen Details werden von der Simulation nicht erfaßt, sind aber für die Darstellungsqualität bei realistisch wirkender Visualisierung von entscheidender Bedeutung. Hier helfen die hinzugerechneten fraktalen Verzerrungen, diesem Ziel nahezukommen.

**3D Texture Iso-Surfaces**

Hierbei wird das untersuchte Skalarfeld auf eine 3D Textur abgebildet (Implementierung siehe Kapitel 7.1.5). Anhand einer bestimmten Transferfunktion und mit dem richtigen Shading werden aus dem Skalarfeld die gewünschten Flächen extrahiert (Abbildung 7.68).

Der große Vorteil dieser Iso-Flächen Darstellung liegt darin, dass sich die Darstellungsgeschwindigkeit bei steigender Isowert/-Flächenanzahl nicht ändert. Beim klassischen MC Verfahren oder auch beim MT Verfahren nimmt die Anzahl der generierten Dreiecke pro zur Darstellung hinzugefügter Iso-Fläche zu. Je mehr Flächen gleichzeitig darzustellen sind, desto mehr Berechnungen fallen an und desto mehr Dreiecksflächen entstehen. Neben der steigenden Belastung der CPU pro zu berechnender Iso-Fläche ist typischeweise die Darstellungsgeschwindigkeit der Graphikkarte direkt abhängig von der Anzahl darzustellender Dreiecke. D.h., je mehr Flächen gleichzeitig auszugeben sind, desto langsamer wird die Visualisierung beim MC und beim MT Verfahren. Ansatzbedingt sinkt diese Geschwindigkeit nicht beim 3D Textur basierten Iso-Flächen Verfahren, d.h. es können beliebig viele Flächen gleichzeitig dargestellt werden, der Berechnungsaufwand für CPU und GPU bleibt konstant. Diese Methode eignet sich folglich besonders gut, wenn viele Isowerte gleichzeitig untersucht werden.

### 7.1.6.2 Ergebnisbilder

(a) (b)

***Abbildung 7.67:*** *Iso-Flächen: (a) Über gewöhnlichen Marching Cubes Algorithmus ermittelt (b) Über Tedraeber basierten Marching Cubes ermittelt.*

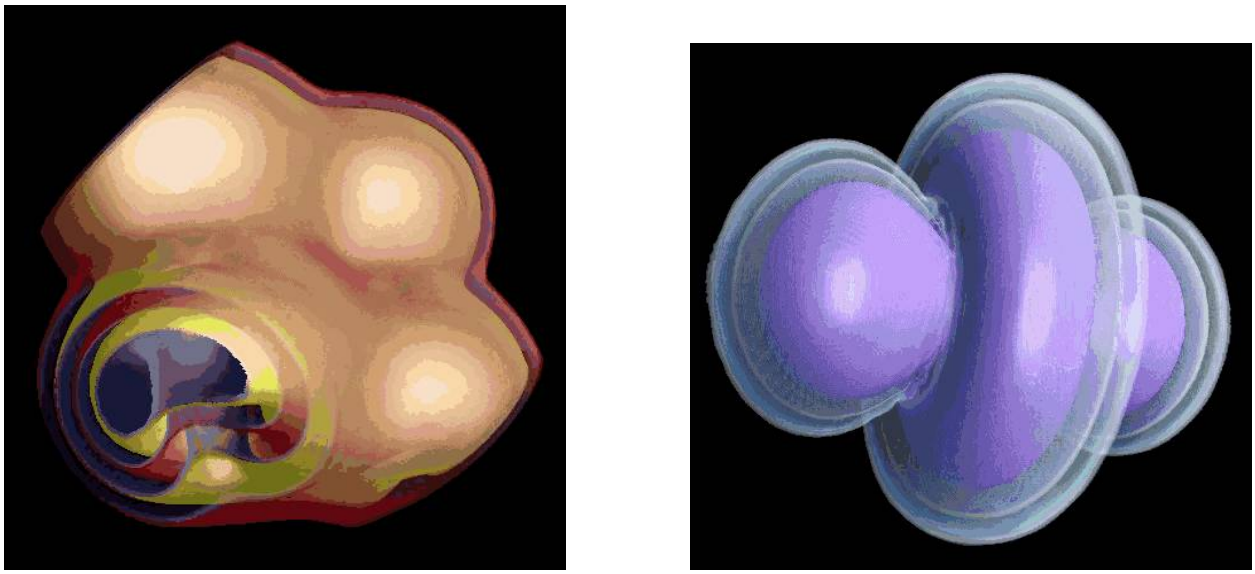


***Abbildung 7.68:*** *Iso-Flächen mit Hilfe von 3D Texturen erstellt.*

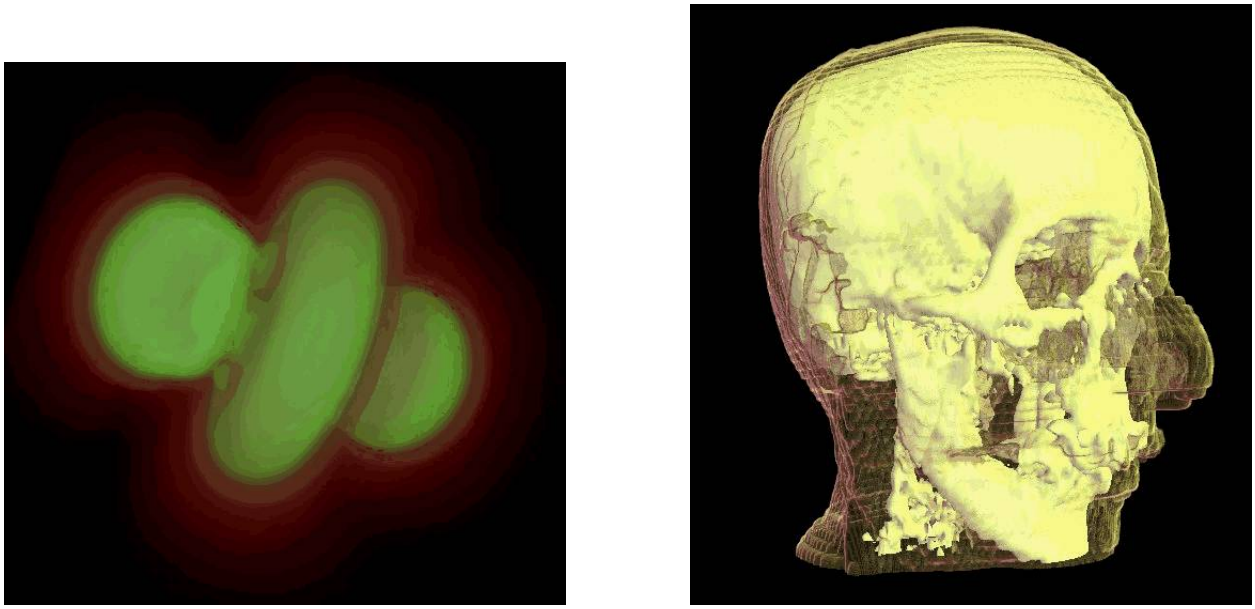


***Abbildung 7.69:*** *Volumen Rendering per 3D Texturen.*

#### 7.1.6.3 Vektorfeld Visualisierungen

| Name | Parallelisierungsart |
|---|---|
| Data Grid | Datenrauma |
| Arrow Plot | Datenraum |
| Line Integral Convolution | Algorithmus |
| Streamlines | Algorithmus |
| Partikel | Algorithmus |

***Tabelle 7.2:*** *Übersicht über die zur Verfügung stehenden Vektorfeld-Visualisierungsmethoden.*

**Data Grid**

Für diese Visualisierung trifft dasselbe zu, wie es auch bei der entsprechenden Methode für Skalarfelder der Fall ist. Lediglich die Ausgabe ist ein wenig angepaßt: Es werden pro Gitterpunkt drei Werte ausgegeben (Geschwindigkeitkomponente der Strömung in $X$, $Y$, und $Z$ Richtung). Die Farbe des dargestellten Gitterpunktes wird anhand des relativen Geschwindigkeitsbetrages bestimmt.

**Arrow Plot**

Bei diesem Verfahren wird ein regelmäßiges Gitter in den Datenraum gelegt. Die Auflösung entlang der Koordinantenachsen der Probe ist pro Achse getrennt regelbar. An die dadurch entstehenden Gitterpunkte wird ein kleiner Geschwindkeitspfeil angebracht, dessen Länge und Orientierung sich nach der dort vorgefundenen Strömung richtet.
Diese Visualisierungsmethode eignet sich ähnlich der Data Grid Visualisierung für den schnellen Überblick über die Strömung. Der Betrachter gewinnt einen ersten groben Überblick über den Gesamtverlauf der Strömung. Jedoch wird die Darstellung aufgrund der Masse der entstehenden Pfeile bei hoher Auflösung des eingestellten Untersuchungsgitters schnell unübersichtlich.

**Line Integral Convolution**

Diese Visualisierungsmethode, deren Implementierung im System in Kapitel 7.1.2 genauer vorgestellt wird, verzerrt eine in den Datenraum gelegte 2 dimensionale (Rausch-) Textur anhand der vorgefundenen Strömung. Hierbei ist die Abtastgenauigkeit für die Verzerrungsberechnung einstellbar. Neben zur Laufzeit generierten Rauschtexturen lassen sich auch beliebige andere Texturen oder Bilder zur Verzerrung einladen. Gitter in unterschiedlicher Anordnung eignen sich auch gut zur Darstellung.
Angewendet wird diese Visualisierungsmethode in erster Linie zur Auffindung von lokalen Wirbeln in turbulenten Strömungen. Da in einer konstanten Strömung alle Abtastpunkte denselben Geschwindigkeitsvektor enthalten, würde in so einem Fall die Textur gleichmäßig verschoben. Im resultierenden Bild wäre nichts Auffälliges zu entdecken. Enthält die Strömung allerdings Wirbel, d.h. Bereiche, in denen sie unterschiedliche Richtungen verfolgt, so wird die Textur an dieser Stelle unterschiedlich verzerrt. Es entstehen „Schlieren“, die schnell zu identifizieren sind. Neben konstanten Strömungen läßt sich mit dieser Methode auch nur schwer der untersuchte Geschwindigkeitsbetrag

analysieren. Nur die Richtung der Strömung bzw. unterschiedliche Richtungen haben einen großen Einfluß auf das Ergebnisbild.

**Streamlines**

Bei dieser Visualisierungsmethode (Implementierung vgl. Kapitel 7.1.3) werden virtuelle „Bänder“ in die Strömung gehängt. Angeknüpft an einen „Emitter“, d.h. einer vorgegeben räumlichen Position, formt die Strömung die Gestalt der Bänder. Deren Anzahl, ihre Färbung, Schattierung und Anordnung ist genau wie die Exaktheit des zugrundeliegenden mathematischen Algorithmus, wählbar (Abbildung 7.70).
Diese Visualisierungsmethode eignet sich gut zur Darstellung des Strömungsverlaufs ausgehend von einem bestimmten Punkt, beispielsweise der Einströmöffnung einer Frischluftzufuhr in einem Auto. Der Verlauf der Bänder gibt an, an welche Stellen im Innenraum die einströmende Luft zirkuliert. Auch ein virtueller Windkanal läßt sich mit Hilfe von Strömungslinien nachbilden. Vor dem untersuchten Gegenstand (beispielsweise eine Tragfläche) werden mehrere dieser Bänder in die Strömung „gehängt“ und deren Verlauf rund um das untersuchte Objekt verfolgt. Wirbel und Turbulenzen lassen sich schnell durch flatternde oder zirkuläre Verläufe erkennen.

**Partikel**

Die Parikel Visualisierung (Implementierung siehe Kapitel 7.1.4) ähnelt der, der Streamlines. Anstatt der einfachen Färbung kommt hier noch die Möglichkeit hinzu, die Größe und Gestalt der Partikel wählen zu können. Über einfache geometrische Formen ist auch die Möglichkeit gegeben, „Billboards“, d.h. flache, dem Betrachter zugewandte Bilder auswählen zu können (Abbildung 7.71).
Prinzipiell eignet sich die Partikeldarstellung für dieselben Einsatzgebiete wie die Streamlines. Auch hier kann gut, ausgehend von einem Einströmpunkt, der Verlauf der untersuchten Strömung verfolgt werden. Neben der reinen technisch-wissenschaftlichen Untersuchung, ist außerdem die realitätsnahe Darstellung mit Hilfe der Partikelvisualisierung interessant. Mit den richtigen Darstellungsparametern und ausreichender Partikelanzahl läßt sich z.B. der Eindruck von Nebelschwaden oder „Wölckchen“ erwecken.

#### 7.1.6.4 Ergebnisbilder

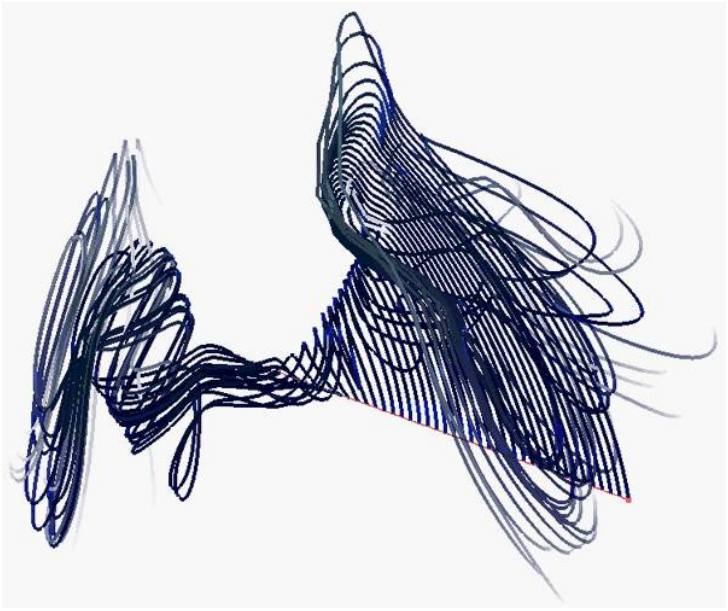


***Abbildung 7.70:*** *Strömungslinienvisualisierung: Streamlines mit Color Mapping.*

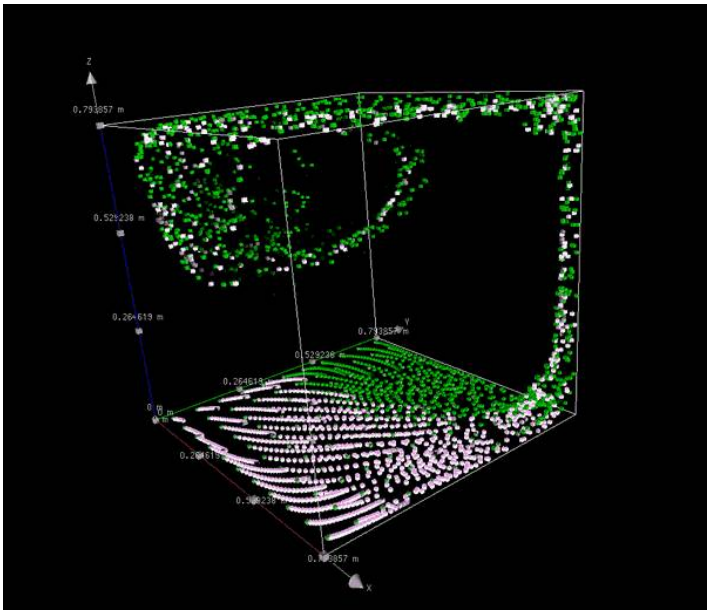


***Abbildung 7.71:*** *Partikelvisualisierung.*

## 7.2 Projekte

Das Visualisierungssystem, das durch diese Arbeit entstand, wurde als Marke unter dem Namen *HereVR*© geschützt (Abbildung 7.72). Die entsprechenden Markenrechte liegen beim Fraunhofer Institut für Graphische Datenverarbeitung.

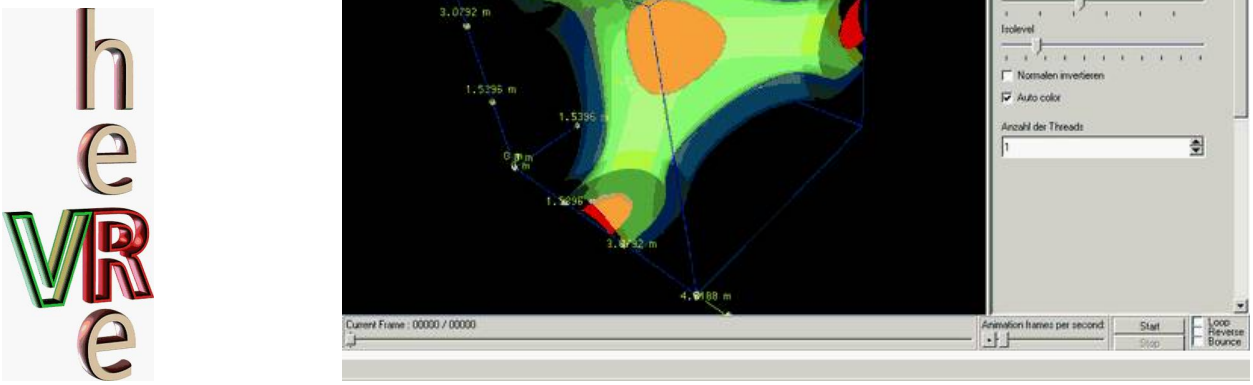



***Abbildung 7.72:*** *Das im Rahmen dieser Arbeit entstandene Visualisierungssystem HereVR*©.

Es integriert die in Kapitel 5 beschriebenen Ideen (mit Ausnahme des dort vorgestellten Datenformates - hierfür wurde ein eigenständiger Prototyp entwickelt) in eine Visualisierungsumgebung und stellt gewissermaßen den funktionalen „Proof of Concept“-Nachweis der hier vorgestellten Ansätze und Konzepte dar. Der zugehörige Aufbau und die Funktionsweise wird in Kapitel 6 beschrieben.

Darüber hinaus wurde und wird das Visualisierungssystem in einer Reihe von Forschungsprojekten des Fraunhofer Instituts für Graphische Datenverarbeitung eingesetzt, die im Anschluss hier kurz vorgestellt werden.

### 7.2.1 COSIWIT

Der Schwerpunkt des Cosiwit Projektes [COS05], gefördert vom BMBF 2002-2004, lag auf dem Bereich gekoppelter Simulationen. Der Großteil der beteiligten Institute und Universitäten in diesem Forschungsprojekt war für die Entwicklung und Bereitstellung von Berechnungsmodellen dieser Simulationen zuständig. Hierbei wurden die unterschiedlichsten Bereiche abgedeckt, angefangen von der Simulation eines Laserschweißgerätes mit Hochdruckgasgebläse bis hin zu einer Ultraschall-Schockwelle, die ein eingespanntes Stück Aluminium verformt. Die Hauptaufgabe des Fraunhofer Instituts für Graphische Datenverarbeitung in diesem Zusammenhang war die Entwicklung eines Visualisierungssystems für die von den Partnern produzierten Datenmassen.

Die Daten für die Visualisierung wurden von den Projektpartnern in unterschiedlichen Datenformaten bereitgestellt (siehe Kapitel 3.5). Nach ihrer Konvertierung in ein vom Visualisierungssystem lesbares Format wurden sie visualisiert. Die Projektpartner bekamen eine lauffähige Version von HereVR zur Verfügung gestellt, mit der sie auf ihren eigenen PCs entsprechende Visualisierungen und Untersuchungen der Simulationsberechnungen durchführen konnten. Die Ergebnisse wurden bei Bedarf teils als Screenshots, teils als animierte Videosequenzen exportiert und so für spätere Analysezwecke konserviert.


***Abbildung 7.73:*** *Struktur des Visualisierungssystems im Cosiwit Projekt.*

Im Rahmen des Cosiwit Projektes wurde HereVR als eigenständige Visualisierungslösung für die Ergebnisdaten der gekoppelten Simulationsrechnungen eingesetzt. Als Szenegraph zur Ausgabe wurde Open Inventor [COI] verwendet. Zielplattform war Windows [Micb] und als GUI Bibliohek kam QT [QT] zum Einsatz (siehe Abbildung 7.73).

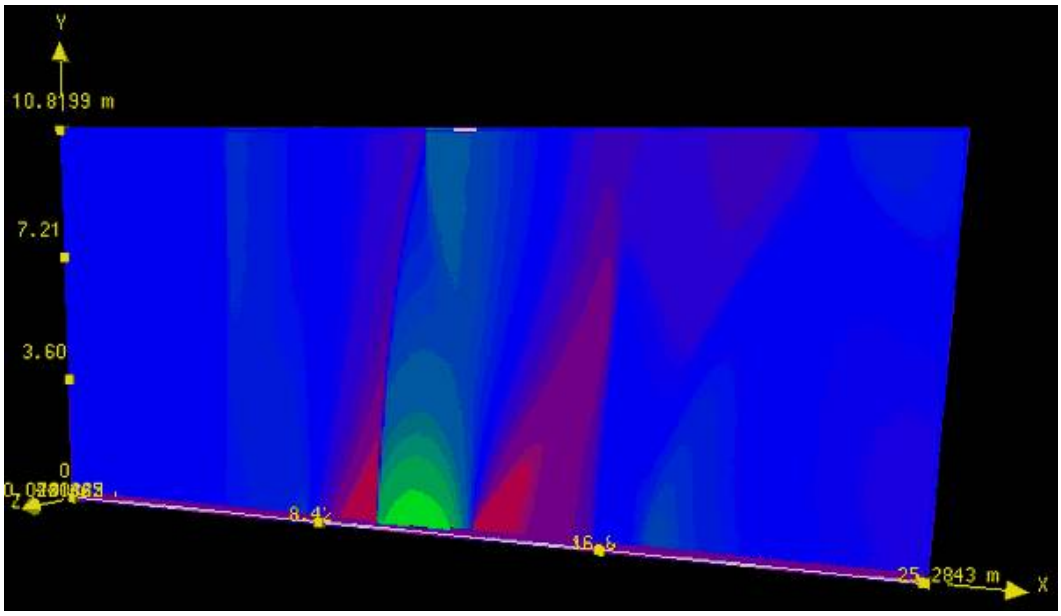



***Abbildung 7.74:*** *Anwendungsfall im Rahmen des Cosiwit Projektes: Eine Schockwelle trifft auf eine eingespannte Aluminiumplatte. Schnittebene durch ein Volumenrendering, 1381 Zeitschritte, 5.8GByte*

.

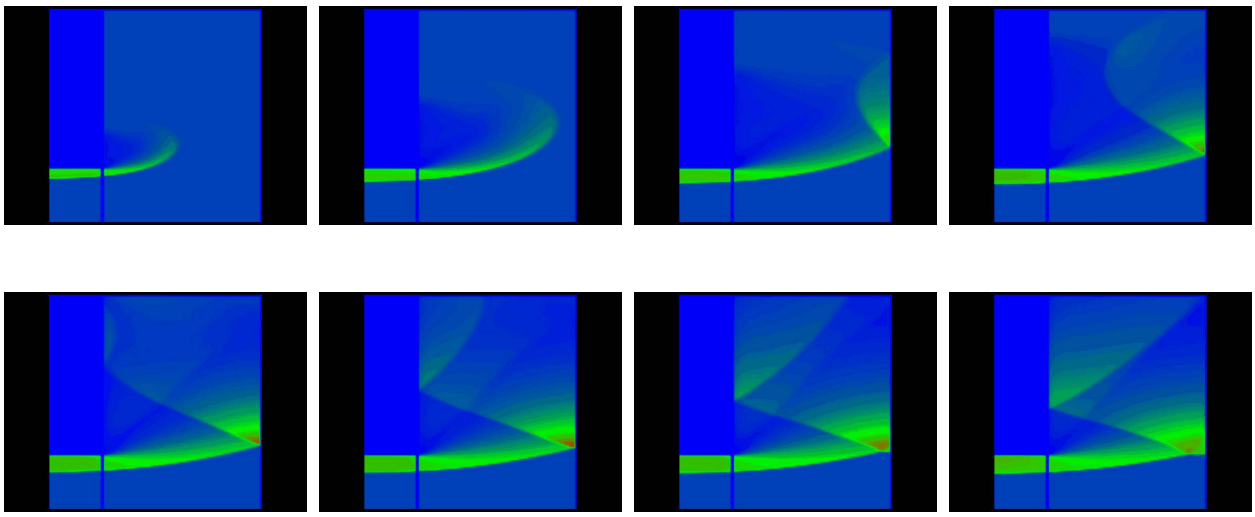


***Abbildung 7.75:*** *Schallwelle, die in ein festes Material übertragen wird. In der Bildfolge zu sehen: Die Ausbreitung im Material und die zugehörigen Interferenzen bzw. Reflexionen am Rand.*

### 7.2.2 VIRTUALFIRES

Der Kern des Virtualfires Projektes [VIR], gefördert von der EU 2002-2004, beschäftigte sich mit der Simulation, Visualisierung und Analyse von Tunnelbränden in bestehenden und geplanten Tunneln. Während sich andere Projektpartner um die Modellbildung und Durchführung der Simulationrechnung kümmerten, war es die Aufgabe des Fraunhofer Instituts für Graphische Datenverarbeitung die zugehörige Visualisierung bereitzustellen. Hierbei wurde neben der rein technisch-wissenschaftlichen Visualisierung der Strömungsdaten auch auf eine realitätsnahe Feuer- und Rauchvisualisierung und die Darstellung der Umgebungsgeometrie Wert gelegt.

Neben einer *Post* Visualisierung vorberechneter Feuersimulationsdaten wurde zusätzlich eine Echtzeit (*Realtime*) Visualisierung umgesetzt, die die Simulationsdaten im Augenblick ihrer Entstehung auf dem Ausgabegerät darstellt. Als Ausgabegerät diente neben dem normalen Monitor auch eine CAVE [CAV], in der der virtuell brennende Tunnel begehbar wurde.

Zum Betreiben der CAVE wurde die Visualisierungsumgebung Covise [COV] verwendet. Um die entwickelten Visualisierungstechniken für die massiv parallele Visualisierung der Strömungsdatenmengen in diesem Zusammenhang nutzen zu können, wurden einige Module des Visualisierungssystems angepaßt. Die GUI wurde durch einen Parser ersetzt, der über eine Schnittstelle zum Rest des Systems angesteuert werden konnte. Die Eingaben des Benutzers auf der Programmoberfläche („Cover“ [COV]) wurden in Kommandos gespeichert und über eine Systemschnittstelle an den Parser übergeben, der sie in Actions (siehe Kapitel 6.5) übersetzte und an den Visualisierungskernel übertrug. Die Ergebnisse der Visualisierungsmethoden wurden in den Systeminternen Szenegraph Open Performer [per05] eingesteuert. Als Datenformat wurde das Covise interne Datenformat verwendet. Die Zielplattform war Linux [LIN] (siehe Abbildung 7.76).

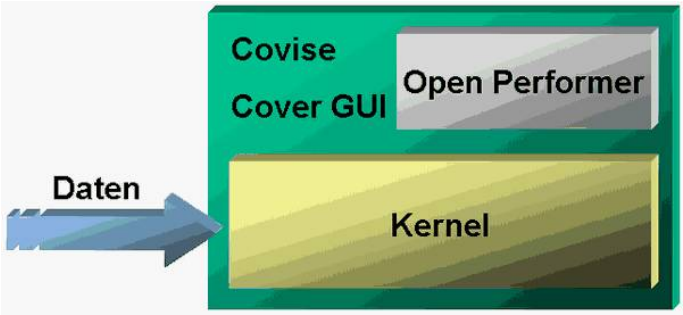



***Abbildung 7.76:*** *Struktur des Visualisierungssystems im Virtualfires Projekt.*

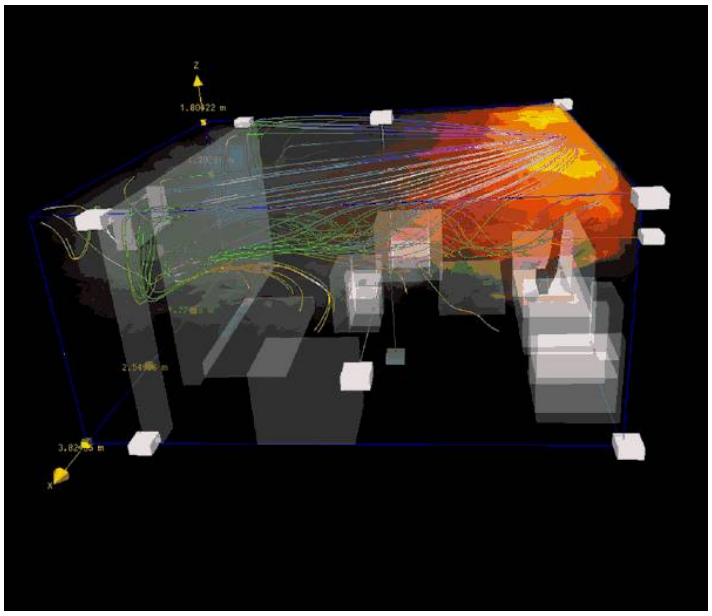


***Abbildung 7.77:*** *Beispieldatensatz des Virtualfires Projektes: Ein simulierter Zimmerbrand. Neben der dargestellten Raumgeometrie, werden die schattierte Strömungslinien und das Fraktale Volumen Rendering zur Darstellung von Feuer + Qualm verwendet.*

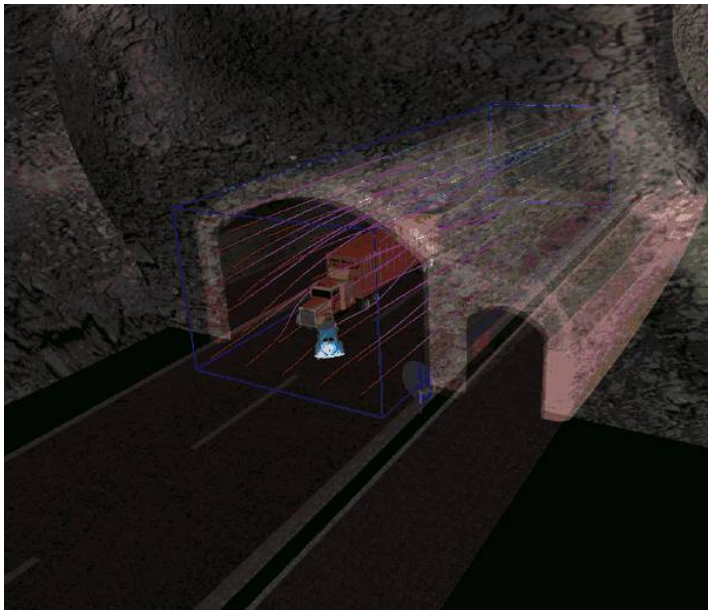


***Abbildung 7.78:*** *Beispieldatensatz des Virtualfires Projektes: Ein simulierter Tunnelbrand nach einer Kollission von 2 Fahrzeugen am Tunneleingang. Die Tunnelgeometrie wird zusammen mit Strömungslinien visualisiert.*

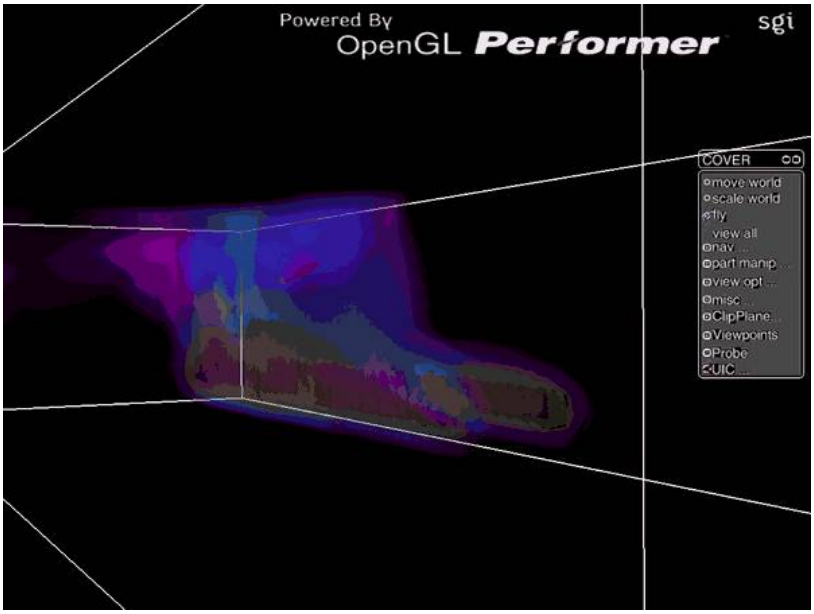



***Abbildung 7.79:*** *Beispieldatensatz des Virtualfires Projektes: Brand in einer U-Bahn Station mit 2 Feuerstellen. Volume-Rendering, Integration in Covise, Betriebssystem Linux.*

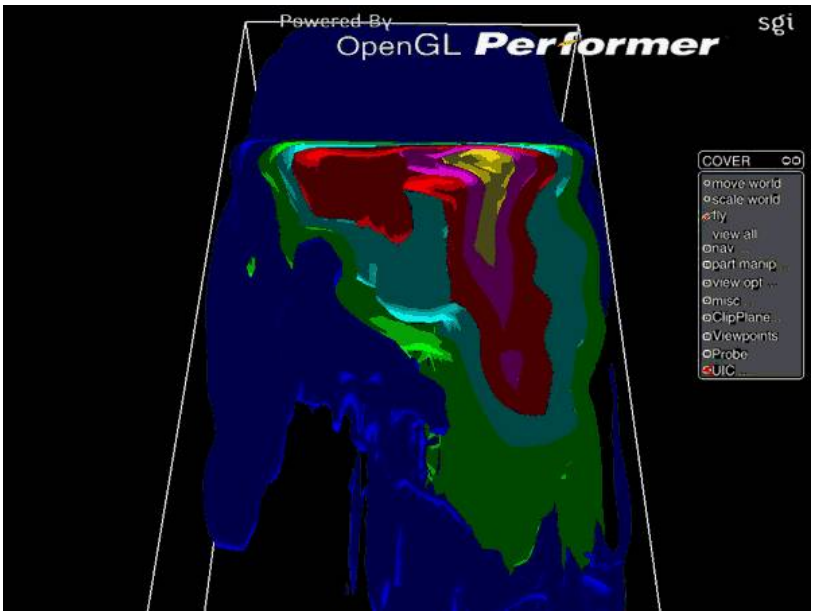



***Abbildung 7.80:*** *Beispieldatensatz des Virtualfires Projektes: Brand im Gleinalm Tunnel mit einer Feuerstelle. Geöffneter Abzugsschacht in der Decke des Tunnels (Blick von oben). Iso-Flächendarstellung mit Marching Cubes, Integration in Covise, Betriebssystem Linux.*

### 7.2.3 UNI-VERSE


Das Uni-Verse Projekt [UNI05], gefördert von der EU 2004-2007, beschäftigt sich mit der Übertragung und kollaborativen Bearbeitung und Visualisierung von Geometrie und Akustikdaten. Eine der zahlreichen Aufgaben, die das Fraunhofer Insitut für Graphische Datenverarbeitung in diesem Projekt übernommen hat, ist die Entwicklung eines Visualisierungssystems für die bereitgestellten Daten - zum einen für einen Low End Client (PocketPC) und zum anderen für einen High End Client (Workstation/PC bzw. Heyewall [HEY]). Der Schwerpunkt liegt hier nicht auf der Anwendung von Visualisierungsmethoden auf Strömungsdaten, sondern auf der Geometrievisualisierung - der Anspruch einer leistungsfähigen parallelisierten Visualisierungslösung bleibt bestehen - sowohl umgesetzt auf PC- als auch auf PDA-Basis. Die zur Visualisierung anstehenden Daten werden über das Netzwerk mit Hilfe des *Verse Protokolls* [VER] kommuniziert.

Die Idee ist es nun, im Rahmen dieses Projektes die Kernkomponenten des in dieser Arbeit entwickelten Visualisierungssystems und den zugehörigen modularen Aufbau zu übernehmen. Die Struktur der einzelnen Module, genau wie ihre parallele asynchrone Arbeitsweise und die actionbasierte Kommmunikation, wird beibehalten, die betroffenen Komponenten jedoch angepaßt. Das Projekt befindet sich zu diesem Zeitpunkt in der Mitte seiner Laufzeit und hat noch viel Entwicklungspotential. In einen ersten Prototypen für den PocketPC und für Heyewall Visualisierung sind bereits die Grundzüge einer entspechenden Adaption von HereVR eingeflossen.

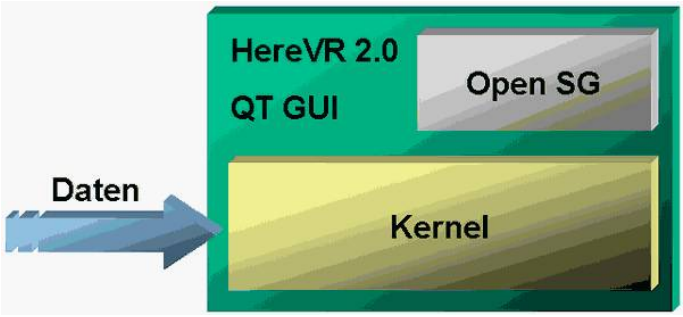



***Abbildung 7.81:*** *Struktur des Visualisierungssystems im Uni-Verse Projekt.*

Für die Übertragung der Funktionalitäten auf den PocketPC und das Ansteuern der High End Visualisierung werden grundsätzlich andere Bibliotheken und Systeme verwendet als in den anderen Projekten zuvor. Auf dem PocketPC ist man auf spezielle Bibliotheken und Entwicklungsumgebungen angewiesen (z.B. die Klimt [kli05] OpenGL Bibliothek für PocketP-

Cs, Embedded C++ [mic05] und noch einige andere), die man für die Umsetzung auf einer PC Plattform nur bedingt verwenden kann, da sie einige Einschnitte und Rahmenbedingungen benötigen (z.B. limitierter Thread und Graphik Support). Für die Umsetzung der High End Visualisierung wird als Ausgabe Szenegraph OpenSG [OPEc] verwendet und ein weiterentwickeltes bzw. angepaßtes Benutzerinterface unter Verwendung von QT [QT] entworfen. Die Zielplattform ist Windows [Micb] für die High End Visualisierung bzw. Windows CE/Mobile [Micc] für den PocketPC (siehe Abbildung 7.81).

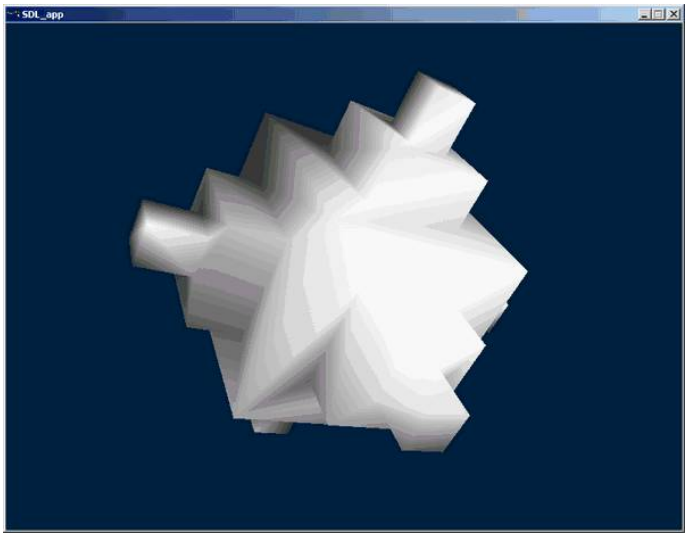



***Abbildung 7.82:*** *Früher Screenshot des Uni-Verse PocketPC Prototypen.*

## 7.3 Zusammenfassung

In diesem Kapitel wird konkret auf ausgewählte Umsetzungen und Beispiele des hier vorgestellten Konzeptes zur massiv parallelen Visualisierung großer Strömungsdatenmengen eingegangen.

Am Anfang des Kapitels werden gezielt einige im Verlauf dieser Arbeit entwickelten Visualisierungsmethoden vorgestellt. Ihre wichtigsten Grundzüge sowie ihre jeweilige Parallelisierung werden erläutert, entsprechende Ergebnisse präsentiert. Dieser Abschnitt gibt eine gute Übersicht, welche Schritte nötig sind, um eine Visualisierungsmethode in das hier vorgestellte System zu integrieren.

Am Ende des Kapitels werden drei besondere Ausprägungen des Systems, die anhand dreier Forschungsprojekte für Visualisierung zustande kamen, kurz vorgestellt. Das in dieser Arbeit entwickelte System wurde prototypisch als Software mit dem Namen HereVR© realisiert und an die jeweiligen Anwendungsfälle angepaßt, sowohl was die Ausgabe als auch die Eingabe betrifft. Der eigentliche Kern des Visualisierungssystems konnte dabei erhalten bleiben, da er wie in Kapitel 4 gefordert, flexibel und portabel angelegt wurde. Verschiedene Beispiele, die den Einsatz des Systems in der Praxis dokumentieren, runden das Kapitel ab.

# Kapitel 8

# Zusammenfassung und Ausblick

Nachfolgend werden die Inhalte der vorliegenden Arbeit anhand der einzelnen Kapitel nochmals kurz zusammengefaßt. Danach folgt eine kurze Übersicht der in dieser Arbeit entwickelten Ergebnisse. Das Kapitel schließt mit einem Ausblick auf mögliche wissenschaftliche Weiterentwicklungen der Inhalte dieser Arbeit.

## 8.1 Inhalte

Ziel der vorliegenden Arbeit war die Spezifikation und Entwicklung eines massiv parallelen Visualisierungssystems für große Strömungsdatenmengen. Dazu wurde zunächst in Kapitel 1 die Problematik sowie die Gründe und Überlegungen vorgestellt, die zur Erstellung der vorliegenden Arbeit beigetragen haben. Im Vordergrund standen hierbei hauptsächlich die großen Datenmengen, die bei physikalisch basierten Strömungsdatenmengen verfahrensbedingt anfallen. Diese Datenmengen stellen anhand ihrer Größe eine Herausforderung an die daran anschließende Visualsierung dar. Durch eine Vereinfachung des Zugangs zur technisch-wissenschaftlichen Visualisierung dieser Datenmengen, sowie einer damit einhergehenden Leisungssteigerung, stehen vielen Anwendern neue Mittel und Wege zur Verfügung, mit Ergebnisdaten aus zugehörigen Simulationen umzugehen. Da man durch Strömungssimulationen auch Szenarien abdecken kann, deren Durchführung in der Realität schwierig bis unmöglich ist, erschließt sich durch das Vorantreiben der zugehörigen Visualisierung ein ständig wachsender Anwenderkreis, der nicht nur rein wissenschaftlich motiviert sein muss. Hierzu gehört beispielsweise die Filmindustrie, die Computerspieleindustrie, als auch die Betreiber von Sicherheitseinrichtungen und Prüfsystemen. Anstatt Abstriche bei der Korrektheit der Simulationsergebnisse zu machen, um die Berechnungen zu beschleunigen, ist es vielversprechender die Visualisierung mit Hilfe der Parallelisierung zu optimieren, .

Diese These konnte in Kapitel 2 anhand verschiedener Anwendungsgebiete nachgewiesen werden. Unterschiedliche Einsatzmöglichkeiten der Visualisierung von Strömungsdaten wurden diskutiert und anhand von Beispielen vorgestellt. Die dabei abgedeckten Szenarien und relevanten Begebenheiten wurden präsentiert und erläutert.

Zur Vorbereitung der Spezifikation eines alternativen Ansatzes, der zur Lösung der vorgenannten Problematik beiträgt, gab Kapitel 3 einen Überblick. Zunächst wurde definiert, was in dieser Arbeit mit dem Begriff Visualisierungsmethode verknüpft wird, gefolgt von einer wissenschaftlichen Einordnung dieser Arbeit. Vor allem die bereits auf dem Markt verfügbaren Visualisierungssysteme für Strömungsdaten sowie die dahinter stehenden Ansätze standen dabei im Zentrum der Analyse. Die zugehörigen Systeme und Methoden wurden kategorisiert und ihre verschiedenen Vor- und Nachteile dargelegt. Es wurde ferner festgestellt, dass alle vorhandenen Ansätze verschiedene Schwachstellen aufweisen, sei es die mangelnde Leistungsfähigkeit der Visualsierung in Anbetracht der darzustellenden Datenmenge, als auch die Abhängigkeit von dedizierten System- bzw. Datenformaten. Erweiterungen der Systeme sowie deren Portierung auf eine andere Plattform bzw. Integration in vorgegebene Softwareumgebungen sind teilweise oder auch gar nicht möglich, Anpassungen an unterschiediche Leistungsstärken verschiedener Endgeräte werden nicht unterstützt. Abgeschlossen wurde das Kapitel mit der Erläuterung allgemeiner und spezieller Grundlagen, die zum genaueren Verständnis dieser Arbeit nötig sind.

Aufbauend auf den vorgestellten Untersuchungen und unter Berücksichtigung der dabei gewonnen Erkenntnisse wurde in Kapitel 4 ein Anforderungskatalog an ein Visualisierungssystem für Strömungsdaten aufgestellt. Hierbei wurden die dafür wichtigen Punkte einzeln aufgelistet und deren Sinn und Zweck erläuert. Unterschieden wurde dabei in Anforderungen, die direkt an das Konzept eines entsprechenden Visualisierungssystems gerichtet sind, und welchen, die sich direkt an die entsprechende technische Umsetzung wenden. Dieses Kapitel dient zur Vorbereitung des anschließenden Konzeptentwurfes.

In Kapitel 5 wurde aus den aufgestellten Anforderungen Stück für Stück ein Konzept für ein massiv paralleles Visualisierungssystem für große Strömungsdatenmengen entwickelt. Jede der zuvor aufgestellten Anforderungen wurde aufgegriffen und fand eine entsprechende Umsetzung. Hierbei wurden für die einzelne Bereiche, wie z.B. Darstellung oder Skalierbarkeit die entsprechenden Lösungsansätze erarbeitet und präsentiert. Abschließend wurden die einzelnen Teile zu einer Gesamt-Systemarchitektur zusammengefügt.

Kapitel 6 beschäftigte sich mit der Spezifikation und Realisierung des vorgestellten Konzeptes. Im Mittelpunkt der Betrachtungen standen hierbei die einzelnen Systemkomponenten und Module, bzw. deren Überführung von der reinen Konzeptidee in konkrete Klassenbeispiele. Nach einer ausführlichen Darstellung der einzelnen Abhängigkeiten und Vererbungsbeziehungen wurden die wichtigsten Funktionen und Parameter in der objektorientierten Programmiersprache C++ erläutert.

In Kapitel 7 wurde schließlich die Umsetzung des Systems anhand einzelner konkreter Beispielen vorgestellt. Hierbei wurde am Anfang des Kapitels auf einzelne Visualisierungsmethoden genauer eingegangen. Diese Visualisierungsmethoden wurden im Zuge der Implementierung aus bekannten Verfahren und Methoden übernommen und speziell dem in dieser Arbeit vorgestellte Visualisierungssystem angepaßt, parallelisiert und weiterentwickelt. Schwerpunkt lag hierbei auf der Verdeutlichung der für die Integration notwendigen Schritte und den zusätzlich durchgeführten Optimierungen bzw. Weiterentwicklungenen der vorgestellten Methoden. Abgeschlossen wurde das Kapitel mit einer kurzen Übersicht über die Forschungsprojekte, in denen das hier entworfene Visualisierungssystem zum Einsatz kam

bzw. immer noch kommt.

## 8.2 Ergebnisse

In der vorliegenden Arbeit wurde ein parallelisierter Ansatz für ein Visualisierungssystem von Strömungsdaten entwickelt. Zentraler Bestandteil war dabei der Begriff Visualisierungsmethode, der zum klaren Verständnis definiert wurde. Die Fähigkeit, möglichst viele Visualisierungsmethoden zu unterstützen ist für ein Strömungsdatenvisualisierungssystem von entscheidender Bedeutung. Sie ist genauso wichtig, wie die Möglichkeit, es durch die ständige Integration weiterer Methoden auf einem modernen und aktuellen Stand zu halten, da die zur Verfügung stehende Technik und die damit verbundenen Möglichkeiten im Computergraphik Bereich einem sehr schnellen Innovationszyklus unterliegen. Hier ist ein weiterer Spielraum für die Integration neuer Techniken und Verfahren entscheidend. So können die aus der Simulation stammenden Datenfelder auf dem Bildschirm visualisiert werden, um sie dem Anwender zur Analyse oder auch zu reinen Präsentations- und Unterhaltungszwecken zur Verfügung zu stellen.

Wichtig für die erfolgreiche Entwicklung eines Visualisierungssystems ist die Integration verschiedenster Ansätze und Verfahren, die die in Kapitel 4 beschriebenen Anforderungen bzw. die dahinter steckenden Probleme lösen können. Viele der bereits auf dem Markt verfügbaren Produkte zu diesem Thema beschäftigen sich meist nur mit Teilaspekten der beschriebenen Problematik. Insbesondere im Bereich der Größe der Datenmenge und der Parallelisierung wurden wichtige Potentiale aufgezeigt und Algorithmen entwickelt, die bis dato in dieser Form noch in keinem anderen System zur Verfügung standen.

Ferner wurde in der vorliegenden Arbeit der Prototyp HereVR©, eine am Fraunhofer Institut für Graphische Datenverarbeitung (IGD) in Darmstadt entwickelte Software, vorgestellt. Hierbei handelt es sich um ein modular aufgebautes parallelisiertes Visualisierungssystem für Strömungsdaten. Durch seinen Aufbau, die interne Struktur und die verwendeten Hilfsmittel und Bibliotheken ist es plattformübergreifend universell als Visualisierungswerkzeug einsetzbar. Es läßt sich sowohl als Komplettsystem betreiben, als auch die benötigten Module herauslösen, anpassen und in vorgegebene andere Visualisierungssysteme integrieren. Mit dieser Software, in die alle Ergebnisse der vorliegenden Arbeit eingeflossen sind, konnte zudem der Beweis erbracht werden, dass das vorgestellte Konzept die gestellten Anforderungen und Erwartungen an ein massiv paralleles Visualisierungssystem für Strömungsdatenmengen erfüllt. Ferner wurde es für zukünftige Erweiterungen und Entwicklungen gerade auch im Hardwarebereich vorbereitet, so daß auch kommende Entwicklungen in dieser Richtung auf absehbare Zeit ihren Weg in das System finden können.

## 8.3 Ausblick

Im Zuge der sich sehr schnell entwickelnden Hardware und den damit ständig steigenden technischen Möglichkeiten ist es wichtig, ein Visualisierungssystem zur Verfügung zu haben,

in das diese neuen Techniken möglichst schnell und auf einfache Art und Weise integrierbar sind. Neben der reinen Steigerung der Leistungsfähigkeit der zur Verfügung stehenden Hardware ändert sich auch gerade im Graphikbereich dadurch ausgelöst sehr schnell die Herangehensweise an bekannte Probleme. Neue Lösungsmethoden und Algorithmen werden durch sich ständig ändernde technische Begebenheiten möglich gemacht. Hier ist insbesondere die Shader-Technologie zu nennen, die im Moment sehr stark nach vorne drängt und ein großes Entwicklungspotential mit sich bringt. Wie bereits an ersten beispielhaften Visualisierungsmethoden (siehe Kapitel 7) gezeigt, ist das hier vorgestellte Visualisierungssystem bereit, genau auf diesem Sektor mit dem Voranschreiten der Technologie Schritt zu halten, um die neuen sich anbietenden Verfahren und Methoden direkt integrieren zu können. Die Integration neuer Visualisierungsmethoden und -verfahren wird auch in Zukunft ein wichtiges Thema für die hier entwickelte Software bleiben. Spannend bleibt das Thema, inwiefern in Zukunft Technologien wie *Real Time Raytraycing* [rea05a, rea05b] die heute üblichen Verfahren zur grafischen Ausgabe ablösen werden und inwieweit sich das auf die heutzutage verfügbaren Visualisierungssysteme auswirken wird. Erste Ansätze zur Realisierung dieser Technologien in Hardware stehen in den Startlöchern (z.B. SaarCOR [saa05, SWS02], OpenRT [ope05], [SWW+04]).

Neben den starken Entwicklungssprüngen im Graphiksektor der vergangenen Jahre, setzt die Hardwareentwicklung im Hauptprozessor Bereich neuerdings neben der Verschaltung mehrerer einzelner Prozessoren auf einem Mainboard auch vermehrt auf Multi-Core Technologie. Hier werden auf einem Prozessor-Element mehrere Prozessor Kerne (CPU Cores) untergebracht, das Mehrprozessor-System also auf einem Chip realisiert. Nach der Einführung der Hyperthreading [hyp06] Technik stellen die modernen *Multi-Core Systeme* für den Einsatz massiv paralleler Visualisierungssysteme und Algorithmen eine prädestinierte Basis dar. Der Sprung von derzeitigen Dual Core / 2-Kern Systemen (z.B. AMD FX 64 X2 [amd06], Intel Core 2 [int06]), zu den angekündigten 4-Kern Systemen bis hin zu multiple Core CPUs verspricht hier ebenso deutliche Geschwindigkeitsgewinne für die Zukunft. Der Graphikmarkt geht mit der Verschaltung mehrerer GPUs auf einer Graphikkarte (Dual GPU [dua06]) bzw. der Bündelung der Rechenkraft mehrerer Graphikkarten (NVidia SLI [nvi06], AMD Crossfire [ati06]) bis hin zu Quad-SLI [dua06] einen ähnlichen Weg in Richtung massive (Hardware-)Parallelisierung.

Adäquate, physikalisch basierte und korrekte Strömungsvisualisierung wandelt sich vom Spezialanwendungsfall zur Massenanwendung (vgl. Kapitel 2). Auch hier ist es wichtig, ein System anzubieten, dass leicht an verschiedene Rahmenbedingungen anpaßbar ist. Die Flexiblität des hier vorgestellten Systems läßt es zu, aus einer wissenschaftlichen Programmoberfläche mit wenigen Schritten (durch Anpassung des GUI Moduls) ein für weniger technisch versierte Anwender nutzbares Werkzeug zur Verfügung zu stellen. Auch hier bietet sich ein weites Feld neuer Einsatzgebiete der vorgestellten Technik an, wie beispielsweise das Erstellen von hochwertigen Präsentationen oder gar die Verwertung in der Film und Computerspiele-Industrie. Hier muss jedoch nicht, wie heute noch üblich, auf „täuschend echt wirkende“ Strömungsdaten für die Visualisierung zurückgegriffen werden, sondern es besteht die Möglichkeit, durch die Leistungssteigerung der Visualisierung in Bezug auf die der Simulation, auf korrekt simulierte Daten zugreifen zu können. Gerade die Abkopplung und Kapselung der Visualisierungsmethoden an sich in ein separates Modul ermöglicht, es für diese Zwecke

technisch-wissenschaftliche und realistische Darstellungen zu mischen.

Das hier vorgestellte System wurde von dem Problem der effektiven und parallelen Ausgabe der fertig berechneten Visualisierungsdaten durch die Nutzung der Szenegraph-Technologie (z.B. OpenSG [OPEc]) weitestgehend abgekoppelt. Allein die Parallelisierung und Optimierung der Vorberechnungen für die nachfolgende Visualisierung verbleibt als Schwerpunkt im Visualisierungssystem. So bleiben auch in Zukunft die mit der an die Vorberechnungen anschließenden parallelen Ausgabe der fertigen Visualisierungsdaten verbundenen Fragestellungen ein separates Thema und können weiterhin unabhängig von dem Schwerpunktthema dieser Arbeit, dem Problem der effizienten und parallelen Bereitstellung und Vorberechnung der eigentlichen Visualisierungsmethoden, betrachtet werden. Im Laufe der Entstehung dieser Arbeit wurde beispielsweise OpenSG um die Lauffähigkeit auf einem Visualisierungscluster erweitert. Diese Neuerung kann durch die strikte Trennung der Berechnungsparallelisierung von der eigentlichen Ausgabe für das Visualisierungssystem übernommen werden.

Ein wichtiger Schwerpunkt dieser Arbeit beschäftigte sich mit der Optimierung des Datenflusses bzw. der Entwicklung eines speziell für die Visualisierung von Strömungsdaten entwickelten progressiven Datenformates. Hier ist ein weiterer wichtiger Forschungsbereich, dessen Weiterentwicklung viel neues Potential verspricht. Gerade für den Bereich der Strömungsdaten existieren bisher noch keine für die Visualisierung optimierten Datenformate, die progressives und/oder sogar gebietsabhängiges Ein- und Auslagern von Datenteilen unterstützen (etwa durch Auswerten der Sichtpyramide des Betrachters zur Festlegung von *Levels of Details* für unterschiedliche Datenbereiche). Die zusätzliche Integration dieser Konzepte, die in Kapitel 5.2.1.1 und 5.4.4 vorgestellt wurden, in das Visualisierungssystem, sowie deren Weiterentwicklung, würde eine weitere Verbesserung der Funktionalität und eine entsprechende Leistungssteigerung mit sich bringen.

Letztendlich bleibt als letzte Ausbaustufe des Systems noch die Verfeinerung des zugrundeliegenden Konzeptes. In diesem Zusammenhang wäre beispielsweise die konsequente abstrakte Beschreibung der Eingabedaten von Visualisierungsmethoden zu nennen. Durch die Einführung einer entsprechenden formalen Beschreibungssprache in das System würde eine weitere Abstraktionsebene eingebracht, die den Umgang mit neu zu entwickelnden und integrierenden Visualisierungsmethoden und den zugehörigen Steuervorgängen bzw. Parametereingabefeldern (Actions - siehe Kapitel 5.3.1.1) vereinfacht. Eine weitere solche Verfeinerung wäre beispielsweise das Einführen einer automatisierten Skalierung des Systems: Anhand von selbständig ablaufenden Messungen könnte die Leistungsfähigkeit der jeweils zur Verfügung stehenden Hardware ermittelt und die zugehörige Visualisierung, bzw. die Granularität der damit verbundenen Parallelisierung entsprechend automatisch angepaßt werden.

Über die Frage nach der Zukunft der Strömungsvisualisierung kann man sich streiten, relativ sicher ist jedoch, dass der Aufwand der damit verbundenen Berechnungen mit der ständigen Leistungssteigerung moderner Computer auch in Zukunft immer weiter zunehmen wird. Strömungssimulationen werden immer präziser, die untersuchten Modelle immer genauer. Das mit der Leistungsteigerung einhergehende Wachstum der zu visualisierenden Datenmenge bleibt dadurch immer eine aktuelle Aufgabe, der sich die Strömungsdatenvisualisierung auch in Zukunft stellen muss. Das hier vorgestellte Konzept bleibt durch seine Verbundenheit mit dieser Thematik also auch weiterhin aktuell.

# Literaturverzeichnis


[11192] ISO/IEC Draft International Standard (DIS) 11172. Information technology - coding of moving pictures and associated audio for digital storage media up to about 1,5 mbit/s (mpeg). *International Organization for Standardization*, 1992.

[3DC] 3d connexion virtual input devices. http://http://www.3dconnexion.com/.

[3dc05] 3dconcept.ch. http://www.3dconcept.ch/artikel/bump, 2005.

[Ake93] K. Akeley. Realityengine graphics. In *Computer Graphics, Proceedings of SIGGRAPH '93*, pages 109–116, 1993.

[Ale01] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley Verlag, 2001.

[amd06] Amd desktop processors. http://www.amdcompare.com/us-en/desktop/, 2006.

[AMI] Amira: Advanced 3d visualization and volume modeling. http://amira.zib.de/index.html.

[ANS85] ANSI. American national standard for information processing systems – computer graphics – graphical kernel system (GKS) functional description. Technical Report ANSI X3.124-1985, American National Standards Institute (ANSI), New York, NY, 1985.

[ANS88] ANSI. American national standard for information processing systems – computer graphics – programmer's hierarchical interactive graphics system (PHIGS) functional description, archive file format, clear-text encoding of archive file. Technical Report ANSI X3.144-1988, American National Standards Institute (ANSI), New York, NY, 1988.

[ati] Ati. http://www.ati.com.

[ati06] Ati crossfire. http://www.ati.com/technology/crossfire/overview.html, 2006.

[AVS] Avs: Advanced visual systems. http://www.avs.com.

[BD99] E. Blayo and L. Debreu. Adaptive mesh refinement for finite difference ocean models: first experiments. *Journal of Physical Oceanography*, 29(6), 1999.

[BDMN79] G. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygard. Simula begin. Studentlitteratur, Lund, Sweden, 1979.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative search. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben79] J.L. Bentley. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[Ben05] P. Benölken. *Effiziente Visualisierungs- und Interaktionsmethoden zur Analyse numerischer Simulationen in virtuellen und erweiterten Realitäten.* 2005.

[BG05] P. Benölken and H. Graf. Direct volume rendering of unstructured grids in a pc based vr environment. In *Journal of WSCG Volume 13 No. 1-3, 2005. Proceedings*, pages 25–32, August 2005.

[BGR01] P. Benölken, H. Graf, and J. Rix. Interactive visualization techniques for a virtual reality based analysis of simulation results. In *Proceedings of SeoulSim*, pages 237–247. Korea Society for Simulation (KSS), August 2001.

[BGS04] P. Benölken, H. Graf, and A. Stork. Texture-based flow visualization in augmented and virtual reality environments. In *International Conference in Central Europe on Computer Graphics, Visualization an Computer Vision (WSCG)*, pages 21–24, August 2004.

[Bha93] D. K. Bhatnagar. Position trackers for head mounted display systems: A survey. Technical Report TR93-010, University of North Carolina, Chapel Hill, NC, 1993.

[BHH00] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware.* ACM, August 2000.

[Bli78] J.F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics*, 1978.

[BO84] J. Berger and M. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal for Computational Physics*, 53:484–512, March 1984.

[BSHS05] F. Bet, N. Stuntz, D. Hänel, and S.D. Sharma. Numerical simulation of ship flow in restricted water. http://www.vug.uni-duisburg.de/~norbert/project/7nsh.html, 2005.

[BSM00] I.N. Bronstein, K.A. Semendjajew, and G. Musiol. *Taschenbuch der Mathematik.* Harri Verlag, 2000.

[BSS00] D. Bartz, B.O. Schneider, and C. Silva. Rendering and visualization in parallel environments. *Siggraph 2000*, Course 13, 2000.

[Bur05] W. Burger. http://webster.fhs-hagenberg.ac.at/staff/burger/lva/nrt4-02/slides/slides04.pdf, 2005.

[But97] D.R. Butenhof. *Programming with Posix(R) Threads.* Addison Wesley, 1997.

[CAV] Cave vt device. http://www.igd.fhg.de/igd-a4/projects45.html.de.

[cfx05] Cfx - computational fluid dynamics software & services. http://www.cfx-germany.com/, 2005.

[cgn05] Cgns cfd format. http://www.cgns.org, 2005.

[CL93] B. Cabal and L.C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 263–270, 1993.

[CMPS97] P. Cigoni, C. Montani, E. Puppo, and R. Scopigno. Multiresolution representation and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4), 1997.

[CN93] T.j. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report Technical Report TR93-027, University of North Carolina, Chapel Hill, 1993.

[COI] Coin. http://www.coin3d.org.

[COS05] Cosiwit - gekoppelte simulationen in wissenschaft und technik. http://www-a3.igd.fhg.de/project.phtml?id=2, 2005.

[COV] Covise: A visualization software with support of virtual reality and collaborative working in networks. http://www.vircinity.com.

[CP92] J.H.E. Cartwright and O. Piro. The dynamics of runge-kutta methods. *School of Mathematical Sciences, Queen Mary and Westfield College, University of London*, 1992.

[Dan] J. Dankert. Numerische methoden, vorlesungsskript. *FH Hamburg.*

[Dan84] W. Dangelmaier. Interaktive Layoutplanung mit INTALA – Konzept und Anwendungsbeispiel. VDI Bericht: Rechnerunterstützte Fabrikplanung 518, VDI, VDI Verlag, Düsseldorf, 1984.

[Dar] D.L. Darmofal. An analysis of 3-d particle path integration algorithm. *Dept. of Aerospace Engineering, University of Michigan.*

[del05] Delphi3d opengl hardware registry. http://www.delphi3d.net/hardware/listreports.php, 2005.

[DIV] Divx: Video kompressionalgorithmus. http://www.divx.com.

[DKC+98] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. In *Proceedings of Eurographics/SIGGRAPH workshop on graphics hardware*, pages 69–76, 1998.

[DN68] O.-J. Dahl and K. Nygard. The Simula 67 common base language. Technical report, Norwegian Computing Centre, 1968.

[doo05] Doom 3. http://www.doom3.com, 2005.

[dP97] A. del Pino. *Konzeption eines Rahmens zur Integration von Parallelrechnern in verteilte virtuelle Umgebungen.* PhD thesis, Universität Darmstadt, Darmstadt, 1997.

[dua06] Wikipedia: Dual gpu, quad sli. http://de.wikipedia.org/wiki/Scalable_Link_Interface, 2006.

[EaSK96] E. Encarnação, W. Straßer, and R. Klein. *Graphische Datenverarbeitung I & II.* Oldenbourg Verlag, 1996.

[Ebe94] D.S. Ebert. *Textureing & Modelling.* Academic Press, 1994.

[EIH00] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A fully scalable graphics architecture. In *Proceedings of SIGGRAPH 2000.* ACM, August 2000.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Eurographics, SIGGRAPH Workshop on Graphics Hardware*, 2001.

[eln05] El niño - das phänomen. http://wdw.prosieben.de/wdw/Natur/Naturgewalten/KlimaKollaps/2_ElNino/, 2005.

[far05] Far cry. http://www.farcry.de/, 2005.

[FEM] Femlab: Environment for modeling and simulating scientific and engineering problems based on partial differential equations. Femlab - http://www.femlab.com.

[FL99] L. Freitag and R. Loy. Adaptive multiresolution visualization of large data sets using a distributed memory octree. In *ACM/IEEE Conference on Supercomputing: Proceedings*, January 1999.

[FLU] Fluent: Computational fluid dynamics simulation software. http://www.fluent.com.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions Computers*, C-21(9), 1972.

[FvDH96] J.D. Foley, S.K. van Dam, A. andFeiner, and J.F. Hughes. *Computer Graphics - Princples and Practice.* Addison-Wesley, 1996.

[gam05a] Gamasutra.com. http://www.gamasutra.com, 2005.

[gam05b] Gamedev.net. http://www.gamedev.net, 2005.

[GB94] M. Gervautz and O. Beltcheva. An approach for object-oriented animation design. Technical Report TR-186-2-94-2, Institut für Computergraphik, Technische Universität Wien, Wien, AU, August 1994.

[GG98] R. Grosso and G. Greiner. Hierarchical meshes for volume data. In *Proceedings Computer Graphics International´98*, pages 761–769, 1998.

[gig97] Exploring gigabyte data sets in real-time. *Siggraph 97, Course 4*, 1997.

[Gla84] A.S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 15–22, 1984.

[GLE97] R. Grosso, Ch. Luerig, and Th. Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In *Proceedings Visualization´97 Conference*, 1997.

[GPR+01] H. Garcke, T. Preußner, M. Rumpf, A. Telea, U. Weikard, and K. van Wijk. A phase field model for continuous clustering on vector fields. *IEEE Transactions on Visualization and Computer Graphics*, pages 230–241, 2001.

[han05] http://www.fugroairborne.com.au/Resources/tn/hanfilter.shtml, 2005.

[Har] D.M. Harrison. Convolution. http://www.upscale.utoronto.ca/GeneralInterest /Harrison/TimeSeries/Convolution.pdf.

[hdr05] Herr der ringe ii: Die zwei türme. http://www.derherrderringe-film.de/, 2005.

[HEY] Heyewall. http://www.heyewall.de/.

[Höh02] B. Höhner. *Parallelisiertes Volumen-Rendering mit 3D Texturen*. PhD thesis, Technische Universität Darmstadt, Darmstadt, December 2002. Supervisors: Schneider, S. and Luckas, V.

[HN00] D.J. Holliday and G.M. Nielson. Progressive volume models for rectilinear data using tetrahedral coons volumes. *Symposium on Visualization*, 2000.

[hoc05] Hochtief ag. http://www.hochtief.de, 2005.

[Hop96] H. Hoppe. Prorgressive meshes. In *SIGGRAPH´96: Proceedings*, 1996.

[HWHJ99] B. Heckel, G. Weber, B. Hamann, and K.I. Joy. Construction of vector field hierarchies. In *Proceedings of IEEE Visualization´99*, pages 19–26, October 1999.

[hyp06] Intel hyperthreading. http://www.intel.com/technology/hyperthread/, 2006.

[IBM] Ibm visualization data explorer. http://www.research.ibm.com/dx/.

[ice05] Ice age. http://www.iceagemovie.com/, 2005.

[Int88] International Organization for Standardization. International standard information processing systems – computer graphics – graphical kernel system for three dimensions (GKS-3D) functional description. Technical Report ISO Document Number 8805:1988(E), American National Standards Institute (ANSI), New York, NY, 1988.

[int06] Intel core 2 duo processor. http://www.intel.com/design/mobile/core/duodocumentation.htı 2006.

[ISH98] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. In *Proceedings of SIGGRAPH '98*, pages 141–150. ACM, August 1998.

[Jor04] T. Jordan. *3D Verse Szenen Visualisierung auf PocketPC mit Netzanbindung.* PhD thesis, Technische Universität Darmstadt, Darmstadt, December 2004. Supervisors: Schneider, S. and Luckas, V.

[Kah76] K. Kahn. An actor-based computer animation language. Artificial Intelligence Working Paper 120, Massachusetts Institute of Technology (MIT), Massachusetts, MA, 1976.

[Ken] D. Kenwright. Visualization algorithms for gigabyte datasets. *MRJ Technology Solutions, NASA Ames Research Center, California.*

[Ken97] D. Kenwright. Siggraph 97 course 32. *MRJ Technology Solutions, NASA Ames Research Center*, 1997.

[Kir02] M. Kirikileonis. Seminararbeit zum thema runge-kutta-verfahren. *Molekulare Biotechnologie, Ruprecht-Karls-Universität, Heidelberg*, 2002.

[kli05] Klimt - opengl für pocketpcs. http://www.studierstube.org/klimt, 2005.

[KMSW01] R. Klein, T. May, S. Schneider, and A. Weber. *Real-Time Fluid Animation by Parallel and Stable Solution Techniques*, volume Festschrift zum 60. Geburtstag von Wolfgang Straßer, ISSN 0946-3852, WSI 2001-20. Wilhelm-Schickard-Institut für Informatik, Tübingen, November 2001.

[KMSW02] R. Klein, T. May, S. Schneider, and A. Weber. *Real-Time Fluid Animation by Parallel and Stable Solution Techniques*, volume Selected Readings in Computer Graphics 2001, ISBN 3-8167-5886-X. INI-GraphicsNet, September 2002.

[Kru90] W. Krueger. The application of transport theory to visualization of 3d scalar data fields. In *Proceedings IEEE Visualization '90*, pages 273–280, 1990.

[KSS96] S. Kleiman, D. Shah, and B. Smaalders. *Programming With Threads.* Prentice-Hall, 1996.

[Lac96] P. Lacroute. Analysis of a parallel volume rendering using shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, 1996.

[LB97] B. Lewis and D.J. Berg. *Multithreaded Programming With Pthreads.* Prentice-Hall, 1997.

[LHJ99] E.C. LaMar, B. Hamann, and K.I. Joy. Multiresolution techniques for interactive texture-based volume visualization. *IEEE Visualization '99*, pages 207–214, 1999.

[LIG] Lightning2: A high-performance display subsystem for pc clusters. http://graphics.stanford.edu/papers/lightning2.

[LIN] Linux - betriebssystem. http://www.linux.com.

[LL94] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. *Computer Graphics, SIGGRAPH '94*, pages 451–457, 1994.

[Lor99] E.J. Lorup. Geoinformationsanalyse - rastermodelle. *Skriptum WS99/00*, 1999.

[map05] Maple. http://www.mapleapps.com/, 2005.

[mat05] Mathlab. http://www.mathlab.com, 2005.

[MDSH] E. Minty, R. Davey, A. Simpson, and D. Henty. Decomposing the potentially parallel. http://www.epcc.ed.ac.uk/computing/training/document_archive/decomp-course/Decomposing.book_1.html.

[Mey87] B. Meyer. Programming as contracting. Report TR-EI-12/CO, Interactive Software Engineering, Goleta, CA, 1987.

[Mey98] S. Meyers. *Effective C++*. Addison-Wesley, 1998.

[MFC] Microsoft foundation classes: Eine sammlung von dialogelementen, die bei visual c++ mitgeliefert werden. http://msdn.microsoft.com.

[MH98] T. Möller and E. Haines. *Real-Time Rendering*. A K Peters Ltd., 1998.

[Mica] Microsoft. Direct 3d. http://www.microsoft.com/windows/directx/.

[Micb] Microsoft. Microsoft windows - betriebssystem. http://www.microsoft.com.

[Micc] Microsoft. Microsoft windows ce - betriebssystem für pocketpcs. http://www.microsoft.com.

[mic05] Microsoft embedded visual c++. http://www.microsoft.com, 2005.

[MOS] Mosix: Cluster software. http://www.mosix.com.

[mpc05] Mpcci - mesh-based parallel code coupling interface. http://www.mpcci.org, 2005.

[MSL02] T. May, S. Schneider, and V. Luckas. Parallel real time fluid simulation and animation with fractal optical refinements. In *Proceedings of the 16th European Simulation Multiconference, Modelling and Simulation 2002*, pages 224–228. ESM, 2002.

[mss05] Shader model 3.0 - no limits. http://www.microsoft.com/whdc/winhec/partners/shadermodel30_NVIDIA.mspx, 2005.

[MSSL03a] T. May, S. Schneider, M. Schmidt, and V. Luckas. Fast scalar- & vectorfield visualization using a new progressive grid class. In *Proceedings of the Simulation and Visualization Conference (SimVis)*, pages 89–101, 2003.

[MSSL03b] T. May, S. Schneider, M. Schmidt, and V. Luckas. The progressive grid: Introducing a new grid class for efficient cfd visualization. In *Proceedings of the High Performance Computing Conference (HPC)*, pages 115–120, 2003.

[MSSL03c] T. May, S. Schneider, M. Schmidt, and V. Luckas. *The Progressive Grid: Introducing a new Grid Class for efficient CFD Visualization*, volume Selected Readings in Computer Graphics 2003, ISBN 3-8167-6608-0. INI-GraphicsNet, 2003.

[MTRB01] H. Maurer, B Thelen, F. Röper, and P. Benölken. Virtueller prüfstand. Technical Report iViP: Leitprojekt integrierte Virtuelle Produktentstehung - Fortschrittsbericht April 2001, Fraunhofer IRB Verlag, April 2001. p141-144.

[MTT84] N. Magnenat-Thalmann and D. Thalmann. CINEMIRA: A 3D computer animation language based on actor and camera datatypes. Technical report, University of Montreal, Montreal, CAN, 1984.

[Mul94] A. Mulder. Human movement tracking technology. Technical report, nserc hand centered studies of human movement project, Simon Fraser University, Burnaby, BC, CAN, July 1994.

[nav05] The engineering toolbox. http://www.engineeringtoolbox.com/21_623qframed.html, 2005.

[nem05] Findet nemo. http://www.disney.de/DisneyKinofilme/nemo/, 2005.

[NK04] Z. Nagy and R. Klein. High-quality silhouette illustration for texture-based volume rendering. In *Journal of WSCG*, pages 301–308, August 2004.

[NMK04] Z. Nagy, G. Müller, and R. Klein. Classification for fourier volume rendering. In *Proceedings of Pacific Graphics 2004*, August 2004.

[NORS97] R. Neubauer, M. Ohlberger, M. Rumpf, and R. Schwörer. Efficient visualization of large-scale data on hierachical meshes. *Visualization in Scientific Computing*, 1997.

[nvi] Nvidia. http://www.nvidia.com.

[nvi06] Nvidia sli. http://de.slizone.com/page/home.html, 2006.

[ogl] Opengl pipeline. http://www.opengl.org/developers/documentation/white_papers/oglGraphSys/section3_4.html.

[ogl05] Opengl shading language. http://www.opengl.org/documentation/oglsl.html, 2005.

[OMN] Omniorb: Eine umsetzung der corba spezification. http://omniorb.sourceforge.net.

[opea] Opengl extension registry. http://oss.sgi.com/projects/ogl-sample/registry/index.html.

[OPEb] Opendx. http://www.opendx.org.

[OPEc] Opensg. http://www.opensg.org.

[ope05] Openrt real-time ray-tracing project. http://www.openrt.de/, 2005.

[OR97] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 59(4):269–285, 1997.

[ozo05] The ozon hole tour. http://www.atm.ch.cam.ac.uk/tour/tour_de/index.html, 2005.

[per05] Open performer szenegraph. http://oss.sgi.com/projects/performer/, 2005.

[plo05] Plot3d - visualization software for computational fluid dynamics. http://www.openchannelfoundation.org/projects/PLOT3D/, 2005.

[POW] Powerflow & powerviz: Cfd simulation & rendering software. http://www.exa.com.

[psv05] Parallel scientific visualization - html link list. http://cs.anu.edu.au/people/Brian.Corrie/research/parvis.html, 2005.

[PV-] Pv-4d. http://www.itwm.fraunhofer.de/.

[pV3] Parallel visual3: co-processing visualization. http://raphael.mit.edu/pv3/pv3.html.

[QT] Qt: Gui library der firma trolltech. http://www.trolltech.com.

[RDHV95] O. Rode, R. Dörner, S. Haas, and D. Vollmer. *Distributed Environment System for Integrated Rendering (DESIRe) – Benutzerhandbuch für Version 7.* Fraunhofer Institut für Graphische Datenverarbeitung (IGD), Darmstadt, December 1995.

[rea05a] Real-time ray-tracing project. http://graphics.cs.uni-sb.de/RTRT/, 2005.

[rea05b] Real-time ray tracing (rtrt). http://www.sci.utah.edu/research/rtrt.html, 2005.

[rid05] The chronicles of riddik. http://www.thechroniclesofriddick.com/, 2005.

[RKT00] S. Roettger, M. Kraus, and Ertl. T. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings Visualization 2000.* IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[Rot03] M. Roth. *Parallel Bildberechnung in einem Netzwerk von Workstations.* 2003.

[RSEB+00] C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[RSEB+01] C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2001.

[RSS96] M. Rumpf, A. Schmidt, and K. Siebert. Functions defining arbitrary meshes, a flexible interface between numerical data and visualization routines. *Computer Graphics Forum*, pages 129–141, 1996.

[SA94] M. Segal and K. Akeley. *The Design of the OpenGL Graphics Interface*. Silicon Graphics Computer Systems, 1994.

[saa05] Saarcor - a hardware architecture for real time ray tracing. http://www.saarcor.de/, 2005.

[Sab88] P. Sabella. A rendering algorithm for visualizing 3d scalar fields. *ACM SIGGRAPH Computer Graphics*, 22(4):51–58, 1988.

[Sah03] J. Sahm. *Kollaborative Visualisierung großer, interaktiver und dynamischer 3D Szenen auf verteilten Endgeräten*. 2003.

[Sak93] G. Sakas. *Fraktale Wolken, virtuelle Flammen*. Springer, Berlin, 1993.

[Sam84] H. Samet. The quadtree and related hierachical data structures. *ACM Computing Surveys (CSUR)*, 16(2), June 1984.

[San92] M. Sannela. The SkyBlue constraint solver. Technical Report TR-92-07-02, Department of Computer Science, University of Washington, Washington, WA, 1992.

[SBGS69] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report Technical Report AFHRL-TR-69-14, NTIS AD700735, U.S. Air Force Human Resources Lab, 1969.

[SC/] Sc/tetra cfd-ware. http://www.sctetra.com/.

[Sch] D.C. Schmitt. Ace: Adaptive coomunication environment. http://www.cs.wustl.edu/schmidt/ACE.html.

[Sch04a] S. Schneider. Interactive parallel visualization of large simulation datasets. *The Colourful World of Computer-Generated Images, Visual Interaction and Visual Communication*, Computer Graphics in Practice, Europäischer Wirtschafts Verlag, 2004.

[Sch04b] S. Schneider. Virtual fires, virtual real time fire emergency. *The Colourful World of Computer-Generated Images, Visual Interaction and Visual Communication*, Computer Graphics in Practice, Europäischer Wirtschafts Verlag, 2004.

[Sch05] S. Schneider. Neon design studio. *Virtual Engineering*, Vol. 02/05:40–41, Feb 2005.

[SE90] B. Stroustrup and M. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Inc., 1990.

[SG05] S. Schneider and Clemens Groß. Neon design studio. *Computer Graphik topics 6/2005*, Vol. 17, 2005.

[sgia] Sgi. http://www.sgi.com.

[SGIb] SGI. Open inventor. http://www.sgi.com/software/inventor/.

[SGIc] SGI. Opengl. http://www.opengl.org.

[SGL01] S. Schneider, C. Groß, and V. Luckas. Prolite 3.0. *Computer Graphik topics*, Vol. 13:29–30, 2001.

[SGS05] S. Schneider, C. Gross, and F. Schmitt. Verfahren zum ermitteln eines zwischenpunktes zwischen zumindest einer ersten randkonturlinie und einer zweiten randkonturlinie, july 2005. Deutsche Patentanmeldung 10 2005 010 370.7, 2005.

[SGS06] S. Schneider, C. Gross, and F. Schmitt. Interaktive konstruktionsverfahren zur zentrierten röhren-/ linienplatzierung in einer vorgegebenen randkontur, sept 2006. Internationale Offenlegungsschrift WO 2006/092189 A2, FPL-Fallnummer 04F45799-IGD, 19.01.2006, Internationales Veröffentlichungsdatum 08.09.2006, PCT/EP2006/000563.

[SH74] I.E. Sutherland and G.W. Hodgman. Reentrant polygon clipping. In *CACM*, pages 32–42, 1974.

[SH05] S. Schneider and M. Hoffmann. Uni-verse. *Computer Graphik topics 6/2005*, Vol. 17, 2005.

[She89] St. R.J. Sheppard. Visual simulation: a user's guide for architects, engineers and planners. *Van Nostrand Reinhold*, 1989.

[shr05] Shrek. http://www.shrek2.com/, 2005.

[SKA+00] S. Schneider, R. Klein, Weber A., T. May, and A. Bryborn. A portable, parallel, real-time animation system for turbulent fluids. In *Proceedings of IASTED Parallel and Distributed Computing and Systems Conference*, pages 394–400. IASTED, March 2000.

[SM00] A. Sanna and B. Montrucchio. Adding a scalar value to 2d vector field visualization: the blic. In *Eurographics 2000*, 2000.

[SM04] S. Schneider and T. May. Realistic visualization of tunnel fires. *The Colourful World of Computer-Generated Images, Visual Interaction and Visual Communication*, Computer Graphics in Practice, Europäischer Wirtschafts Verlag, 2004.

[SMS02] S. Schneider, T. May, and M. Schmidt. Virtualfires. *Computer Graphik topics 6/2002*, Vol. 14, 2002.

[SMS03a] S. Schneider, T. May, and M. Schmidt. Parallel architecture of an interactive scientific visualization system for large datasets. In *Proceedings of the OpenSG Symposium*, 2003.

[SMS03b] S. Schneider, T. May, and M. Schmidt. Rendering large (volume) datasets: A new parallel visualization system. In *Journal of Winter School of Computer Graphics Conference (WSCG)*, pages 418–424, 2003.

[SMS04a] S. Schneider, T. May, and Cosiwit Staff. *Leben und Arbeiten in einer vernetzten Welt*, volume Cosiwit Hochglanzbericht / Infobroschüre. Fraunhofer-IuK-Gruppe, 2004.

[SMS04b] S. Schneider, T. May, and Virtualfires Staff. *Virtualfires: Final Report.* 2004.

[SRBE99] M. Schulz, F. Reck, W. Bartelheimer, and T. Ertl. Interactive visualization of fluid dynamics simulations in locally refined caretsian grids. In *IEEE Visualization 99, Proceedings*, pages 413–553, 1999.

[SSM02] S. Schneider, M. Schmidt, and T. May. Cosiwit. *Computer Graphik topics 6/2002*, Vol. 14, 2002.

[Sta99] J. Stahm. Stable fluids. *ACM Computer Graphics*, Proc. SIGGRAPH ´99:121–128, 1999.

[Sti99] P. Stingl. *Mathematik für Fachhochschulen, 6. Auflage.* Carl Hanser Verlag, 1999.

[Str86] B. Stroustrup. An overview of c++. *SIGPLAN notices*, 21(10), 1986.

[Str91] B. Stroustrup. *The C++ Programming Language, 2nd Edition.* Addison-Wesley Publishing Company, Inc., 1991.

[SvWHP94] A. Sadarjoen, T. van Walsum, A.J.S. Hin, and F.H. Post. Particle tracing algorithms for 3d curvilinear grids. *Faculty of Technical Mathematics and Informatics, Delft University of Technology*, 1994.

[SWS02] J. Schmittler, I. Wald, and P. Slusallek. Saarcor - a hardware architekture for ray tracing. In *Proceedings of EUROGRAPHICS Graphics Hardware 2002*, 2002.

[SWW+04] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of Graphics Hardware*, 2004.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[tda05] The day after tomorrow. http://www.thedayaftertomorrow.com/, 2005.

[TEC] Tecplot. http://www.tecplot.com.

[tec05] Tecchannel.de. http://www.tecchannel.de, 2005.

[Tre02a] J. Treschau. *Interaktive parallele Visualisierung wissenschaftlicher Daten*. PhD thesis, Technische Universität Darmstadt, Darmstadt, January 2002. Supervisors: Schneider, S. and Luckas, V.

[Tre02b] J. Treschau. Interaktive parallele visualisierung wissenschaftlicher daten. Master's thesis, Technische Universität Darmstadt, Darmstadt, January 2002. Supervisors: Schneider, S. and Luckas, V.

[ts105] (t)raumschiff surprise. http://www.periode1.de, 2005.

[twe05] Tweak3d.net. http://www.tweak3d.net/articles/bumpmapping/3.shtml, 2005.

[UNI] Unix - betriebssystem. http://www.unix.com.

[UNI05] Uni-verse. http://www.uni-verse.org, 2005.

[unr05] Unreal 2. http://www.unreal2.com, 2005.

[VER] Verse netzwerkprotokoll. http://www.blender.org/modules/verse.

[ver01] Advanced character physics, verlet integration. http://www.teknikus.dk/tj/gdc2001.htm, 2001.

[VGK96] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. *IEEE Volume Visualization Symposium*, pages 23–30, 1996.

[VIR] Virtualfires: Virtual real time fire emergency simulator. http://www.virtualfires.org.

[VISa] Vis5d: Ssec visualization project. http://www.ssec.wisc.edu/~billh/vis.html.

[VISb] Visicade. http://www.visicade.de.

[vol05] 20th century fox. http://www.foxdeutschland.com/kino/, 2005.

[VSSB95] J. Van Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware '95*, pages 51–55, 1995.

[VTK] Vtk: Visualization toolkit. http://public.kitware.com.

[wal06] N24: Waldbrände in portugal. http://www.n24.de/boulevard/nus/index.php/n2005080514111800002, 2006.

[WE98] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics, Proceedings of SIGGRAPH*, pages 169–177, Orlando, FL, August 1998.

[wet05a] Wetter.com. http://www.wetter.com, 2005.

[wet05b] Wettervorhersage. http://www.jostjahn.de/wolklei6.html, 2005.

[wet06] Nasa: Weather watch. http://www.nasa.gov, 2006.

[WIR] Wiregl: A scalable graphics system for clusters. http://graphics.stanford.edu/papers/wiregl.

[WJS99] R.S. Wright Jr. and M. Sweet. *OpenGL Superbible*. The Waite Group Press, 1999.

[WKL+01] G.H. Weber, O. Kreylos, T.J. Ligocki, J.M. Shalf, H. Hagen, B. Hamann, K.I. Joy, and K.-L. Ma. High-quality volume rendering of adative mesh refinenement data. In *Proceedings of the 6th International Fall Workshop Vision, Modeling and Visualization*, pages 21–23, 2001.

[WM92] P.L. Williams and N.L. Max. A volume density optical model. *Workshop on Volume Visualization*, pages 61–68, 1992.

[WvG92] J. Wilhelms and A. von Gelder. Octrees for faster isosurface generation. *ACM Transactions of Graphics (TOG)*, 11(3), July 1992.

[ZMFM97] Z. Zhu, R. Machiraju, B. Fry, and R. Moorhead. Wavelet-based multiresoultional representation of computational field simulation datasets. In *Proceedings of the Conference on Visualization*, October 1997.

[ZSH] M. Zöckler, D. Stalling, and H.-C. Hege. Interactive visualization of 3d-vector fields using illuminated stream lines. *Konrad-Zuse-Zentrum für Informationstechnik Berlin*.

[Zwa03] M. Zwatschek. *Parallelisierte Partikelvisualisierung*. PhD thesis, Fachhochschule Darmstadt, Darmstadt, December 2003. Supervisors: Schneider, S.

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Index

# Anhang A

# Betreute Diplom-, Bachelorarbeiten und Projektpraktika

## A.1 2002

- J. Treschau: *Interaktive parallele Visualisierung wissenschaftlicher Daten.* Technische Universität Darmstadt (TUD), Diplomarbeit Fachbereich Informatik (FB 20), Fachgebiet Graphisch-Interaktive Systeme (GRIS), Darmstadt, April 2002.

- B. Höhner: *Parallelisiertes Volumen-Rendering mit 3D Texturen.* Technische Universität Darmstadt (TUD), Diplomarbeit Fachbereich Informatik (FB 20), Fachgebiet Graphisch-Interaktive Systeme (GRIS), Darmstadt, Dezember 2002.

## A.2 2003

- M. Zwatschek: *Parallelisierte Partikelvisualisierung.* Fachhochschule Darmstadt (FHD), Diplomarbeit Fachbereich Informatik (FB20), Fachgebiet Graphisch-Interaktive Systeme (GRIS), Darmstadt, August 2003.

- S. Ocko: *Integration von Maßeinheiten, Skalen und Koordinaten in ein System zur Strömungsdatenvisualisierung.* Fachhochschule Darmstadt (FHD), Bachelorarbeit Fachbereich Informatik, Darmstadt, Juli 2003.

- M. Kuhn: *Originaldatenbrowser für diskrete Strömungsgitter Daten.* IHK Darmstadt, Projektpraktikum IT Fachinformatiker Ausbildung, Abschlußprojekt, Fachgebiet Anwendungsentwicklung, Darmstadt, November 2003.

- A. Adolf: *Entwurf und Implementierung eines 2D MAP Editors zur Herstellung von Objekt-Konnektivitäten.* IHK Darmstadt, Projektpraktikum IT Fachinformatiker Ausbildung, Abschlußprojekt, Fachgebiet Anwendungsentwicklung, Darmstadt, November 2003.

## A.3 2004

- T. Jordan: *3D Verse Szenen Visualisierung auf PocketPC mit Netzanbindung.* Technische Universität Darmstadt (TUD), Diplomarbeit Fachbereich Informatik (FB 20), Fachgebiet Graphisch-Interaktive Systeme (GRIS), Darmstadt, November 2004.

- A. Krug: *Visualisierung von Störmungsdaten mit Hilfe eines verbesserten Stream Line Verfahrens.* Fachhochschule Darmstadt (FHD), Bachelorarbeit Fachbereich Informatik, Darmstadt, September 2004.

- F. Schmitt: *Visualisierung von Störmungsdaten mit Hilfe verbesserter Line Integral Convolution (LIC) Verfahren.* Fachhochschule Darmstadt (FHD), Bachelorarbeit Fachbereich Informatik, Darmstadt, September 2004.

- S. Lorsbach: *Integration der Darstellung von texturierten OBJ (Alias Wavefront Geometrie) Dateien in ein Visualisierungssystem.* Fachhochschule Darmstadt (FHD), Bachelorarbeit Fachbereich Informatik, Darmstadt, September 2004.

- M. Kneissl: *Integration multivariater Daten Visualisierung von Geschwindigkeitsbeträgen / -richtungen in ein Visualisierungssystem.* Fachhochschule Darmstadt (FHD), Bachelorarbeit Fachbereich Informatik, Darmstadt, September 2004.

# Anhang B

# Lebenslauf

| | |
|---|---|
| Name: | Sascha Heinz Schneider |
| Geburtstag: | 20. Juli 1973 |
| Geburtsort: | Darmstadt |

**Schulbildung:**

| | |
|---|---|
| 1980 – 1984 | Grundschule,<br>Frankensteinschule Mühltal/Nieder-Beerbach |
| 1984 – 1986 | Förderstufe,<br>Pfaffenbergschule Mühltal/Nieder-Ramstadt |
| 1986 – 1993 | Gymnasium,<br>Lichtenbergschule Darmstadt<br>(Abschluß Abitur) |

**Studium:**

| | |
|---|---|
| Oktober 1993 – Januar 2000 | Mathematik mit Schwerpunkt Informatik,<br>Technische Universität Darmstadt (TUD),<br>(Abschluß Diplom-Mathematiker) |

**Berufspraxis:**

| | |
|---|---|
| 1997 – 1999 | Selbständig,<br>Firma „M@n in Web“ GbR,<br>Internetpublishing,<br>Mühltal/Nieder-Beerbach |
| Februar 2000 – Januar 2006 | Wissenschaftlicher Mitarbeiter<br>Fraunhofer Institut für Graphische Datenverarbeitung (IGD),<br>Abteilung Animation & Bildkommunikation (A3),<br>Darmstadt |

| | |
|---|---|
| Februar 2006 - Juni 2006 | Software-Architekt<br>T-Systems, Systems Integration,<br>Industry Business Unit Telco, CRM 31<br>Darmstadt |
| Seit Juli 2006 | Software-Entwickler<br>GCS / Dentale CAD/CAM Systeme,<br>SIRONA Dental Systems GmbH,<br>Bensheim |